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

© Copyright 2020 by Michael Mattocks

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

List of Tables

List of Figures

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 **??**, **??**, and **??**, I have used the apellation "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 **??**.

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 use 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 [**?**].

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 **??** and **??** 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 **??** and **??** 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 **??**, **??**, 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.

?? 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. **??** 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.

?? discusses the implications of **??** for models of RPCs generally, and in the CMZ particularly; it also recommends a better model selection approach from Bayesian theory, nested sampling. **??** 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.

?? 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 **??** 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 [**?**]. 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

[1] This was in no sense a problem with Harris' understanding. A similarly high-level review coauthored by Pam Raymond [**?**] 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 the retina[2]. 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 [**?**]; Harris' group confirmed this result in *Xenopus* the subsequent year [**?**], 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

[2]Indeed, 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. [**?**]

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. [**?**]

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 steretypical 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 [**?**]. 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 [**?**].

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 "morphogenetic 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 [**?**]. 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 [**?**], which she describes as the "morphogenetic alphabet", I have categorised RPC phenomena as relating to proliferation, specification, migration, growth, death, and extracellular matrix formation[3]. 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 [**?**][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) [**?**, **?**], 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 [**?**]). 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" [**?**]. 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 conventional[5] 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

[3] I 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.

[4] While 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.

[5] Unfortunately, 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. [**?**]
>
> [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 fashion[6] [**?**].

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 [**?**, **?**]. After the central retina is specified, the RPC cell cycle once again lengthens, and is presumed to continue to slow until RPCs have completed differentiation[7].

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[**?**]. 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[**?**].

1.3.2 Specificative phenomena

Irrespective of their location in the retina, vertebrate RPC lineages have offsping 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 [**?**, **?**, **?**]. 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 [**?**]. In general, then, RPCs are taken to be totipotent with respect to the neural retina- all of the cell types[8] 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 (many pioneered by Harris), have broadly confirmed the findings of the 80s and 90s in mammalian fixed specimens, explants, and the like [**?**].

[6] Since 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.

[7] This presumption is demonstrably incorrect in the /textitD. rerio eye, see **??**

[8] The "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.

Of particular note here are the observations of the Raff group [?, ?], 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 [?] by [?]. 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 history[9], vertebrate retinal development displays a temporal ordering, such that in any particular location, retinal ganglion cells (RGCs) tend to be produced first, followed by the other cell types in what has been described as an overlapping "histogenetic order", pictured in Figure ??. 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 pattern[10]. This naturally

[9]Even 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[?], giving lie to the notion that the process is particularly strict.

[10]In many studies, cell cycle exit is conflated with specification such that evidence of the former is taken as evidence

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 specificative programs if they begin to execute it along the spatiotemporal gradient noted above, as originally suggested by Cepko et al. [**?**]. However, it is impossible to reconcile this notion with the recent results of Harris' *in vivo* zebrafish lineage tracing studies [**?**, **?**, **?**], which confirm mammalian cell culture work in demonstrating that vertebrate RPC lineages do not execute identical (or even vaguely similar) specificative 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 ennumerates 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 material[11], 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[**?**]. The environment provided by the apical retinal surface appears to be required for RPC mitosis to proceed[12], 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 [**?**]. This "random walk" may arise from displacements due to the directed INM of neighbouring progenitors [**?**]. More committed RPCs also appear to actively migrate to positions appropriate for the specified cell type [**?**, **?**], 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 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 [**?**], 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

of the latter. There are, however, reasons to believe cell cycle exit and specification are not the same process, discussed below.

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

[12] Nonapical divisions do occur, notably in specified, but proliferative cells [**?**]. 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.

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 [**?**]. 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 [**?**]), 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 [**?**, 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 [**?**]. Pax6 was thus taken to provide a molecular pointer to a potential common ancestor for all animal eyes [**?**].

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 [**?**]. 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 [**?**]. The expansion of this explanation to include other TFs in a network revealed significant differences between species [**?**]- 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) [**?**]. 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 neurons[13]. Moreover, the expansion of the monocausal "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 specification[14], 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 [**?**], and is known to directly activate Ath5, necessary for RGC specification [**?**]. 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 microopthalmia in the mouse [**?**]. Chx10 was subsequently found to repress Mitf, which is involved in RPE specification, and hence to promote neural retinal fates over pigmented epithelial ones [**?**]; 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 [**?**] and the regulation of Vsx1 (a paralogue of Chx10), Foxn4, and Ath5, associated with specification of subpopulations of bipolar cells, horizontal and amacrine cells, and RGCs and PRs, respectively [**?**, **?**].

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 [**?**], 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.

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 [**?**]. These include BMP, CNTF, FGF, Glucagon, Hedgehog, IGF, Notch, TGFα, TGFβ, VEGF, wnt, and a host of neurotransmitters. This diversity of signalling

[13]Indeed, 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 [**?**]

[14]This 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 [**?**, **?**] this terminology should probably be jettisoned.

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)[**?**], 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 [**?**]. Inhibition of FGFR2 signalling, and activation of ephrinB1 signalling have also been implicated in early *Xenopus* eye field cell movements [**?**]. 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 [**?**]. 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 [**?**].

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 [**?**], 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 [**?**], 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 [**?**]. This process is dependent on, and downstream of, the above-mentioned FGF induction [**?**]. The role of Hh signalling has been challenged on the basis that Hh inhibition in subsequent experiments did not display the same effect size [**?**], and that its effects on Ath5 expression, required for RGC specification, are ambiguous [**?**]. 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 [**?**, **?**].

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 [**?**, **?**]. 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 [**?**]. 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 [**?**]. This code seems to be established, in part by the action of Gdf6a, in RPCs themselves, prior to specification [**?**].

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) [**?**]. 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 [**?**].

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 [**?**, **?**]. 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 [**?**]. In mouse retinal explants, pharmacological inhibition of HDAC results in decreased proliferation and specification [**?**]. 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) [**?**]. A switch to other subunits seems to be necessary for specification [**?**]. 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 idiosyncracies, in the hope that doing so will eventually bridge the explanatory "islands" of the MEx archipelago. I have referred to this as the Ennumerationist 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 [**?**, p.22-9]. Moreover, "even as the introduction of greater detail can dissolve order, so the neglect of detail can generate it." [**?**, 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. [**?**, 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 it[15]. 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 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[**?**, **?**, **?**].

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

[15] At least part of this problem is likely related to the fact that the majority of biomedical findings cannot be replicated [**?**]. 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.

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 practicioner'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 **??**.

1.6 Explanatory Strategy and Intent of the SMME

As we have seen in **??**, 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" [**?**]. 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 [**?**, **?**, **?**]. 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 [**?**]. He et al. [**?**] and Wan et al. [**?**] use essentially the same model and make up the substance of the first of these two maneuvers. Boije et al. [**?**] substantially modifies this model, specifying the activity of two known transcription facts (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[**?**]. 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" [**?**][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 ennumerated 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.

[16]This 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[**?**]. 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[**?**, **?**]. 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."[**?**] 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 specificative 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"[**?**]; (2) to have established the functional equivalency of embryonic RPCs and their descendants in the postembryonic circumferential marginal zone (CMZ)[**?**]; and (3) to have established the involvement of causally independent transcription factor signals in the production of unpredictably variable RPC lineage outcomes[**?**]. 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[**?**]. 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[**?**].

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[**?**]. We explored the structure of the SSMs, dubbed the He and Boije SSM respectively, compared to their explanatory forebear, dubbed the Gomes SSM[**?**]. 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 actvity, 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.[**?**, **?**]; 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.[**?**]; 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.[**?**]. 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[**?**]. 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[**?**]. While Fagan treats SSMs seperately 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[**?**].

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"[**?**, **?**]. 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 **??**. 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[**?**]. 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"[**?**], 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.

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

The He SSM, shown in Fig **??**, 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

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.

"causative hypothes[e]s". 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.

The He model is, therefore, called on to explain the temporal structure of the proliferative and specificative 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

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

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 diplayed in Fig **??**. 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 Ptf1a, 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 Ptf1a 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"[**?**] (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[**?**, **?**], nor that using the Boije SSM[**?**] 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, foregoing 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 idiosyncracies. 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 **??**). 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[**?**] 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 Akiake'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 ± 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 **??**, with the output of the two models being presented separately in **??** and **??**. 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 **??**. 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: Akiake'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[**?**], 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 **??**. 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[**?**], 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 μm.

Indeed, the SMME SSMs were originally brought to our attention when the He SSM was deployed in Wan et al.[**?**], 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[**?**], 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 **??**, 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 occuring per lineage, per hour, we obtain the values presented in Fig **??**.

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 ± 95% CI.

Similarly, when we performed cumulative thymidine analogue labelling of the 3dpf CMZ, (using the assumptions of Nowakowski et al. [**?**], 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 **??**. 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[**?**], 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 **??**, 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[**?**]. 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 from the observed 3dpf population distribution, and given an additional $1/10^{\text{th}}$ 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 ± 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[**?**]. Motivated by the observation that momentary mRNA transcript expression in RPCs is highly variable from cell-to-cell[**?**], 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[**?**], 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 Ptf1a 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."[**?**]

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 **??**: has Harris succeeded in finding order by blurring out the chaotic welter of mechanistic explanations for RPC function in retinogenesis? **??** 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. [**?**]. 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 [**?**] 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 **??**, the linear progression of competencies. Because the SSM model form does not usually have spatial dimensions, we did not test models involving variablility 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 inferences[1], 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 **??**, 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 intrinsic[2] [**?**]. 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 [**?**]. 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 are largely determined by intrinsic

[1] Having 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.

[2] Co-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.

processes seemed a striking confirmation of Williams and Goldwitz's much earlier suggestion [**?**], 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 [**?**], 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 [**?**]. 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 [**?**, **?**]. 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 computed[3]. 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 **??** 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 about RGC specification in order to find better models of RPC function. Carefully reconsidering the hypothesis

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

of Neumann et al. [**?**], 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 Akiake'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" [**?**]. 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 optimum[4]. If we are evaluating hypotheses in order to make decisions about potentially years-long

[4]This is part of what is meant by "the curse of dimensionality", when speaking of the difficulty of sampling the loss

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 [**?**], this use of this system has become widespread in cosmology [**?**], 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 status[5]. 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 the hypotheses to be tested, the differentiated central retina may be mostly irrelevant, so the modelled

function in high dimensional parameter spaces.

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

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 eye[6].

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 hte 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 of CMZ activity across the life of the organism, very few of the relevant macromolecular processes are

[6]The 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.

[7]It 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.

[8]This is implemented in CHASTE as "ghost nodes".

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 [**?**]. 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 **??** of the Supplementary Materials, while more extensive theoretical background is available in **??** 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 [**?**] 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 the one that they are simply normally distributed. These results are displayed in Table **??**

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 meaurements, or the estimated whole-annulus population, calculated as described in **??**, even though this quantity is calculated using the normally-distributed lens diameter[1]. 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 are slightly

[1]The 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 (†)	-712.256	**-711.766**	0.491

$\mathcal{N}$: 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 14μm cryosection. CMZ annular pop: population of annular CMZ. RPE: retinal pigmented epithelium. CR: cellular retina.

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 **??**, 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 **??**.

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

Parameter	**$\mathcal{N}$ logZ**	**Log-$\mathcal{N}$ logZ**	**logZR**	**σ significance**
CMZ annular pop.	-744.649 ± 0.012	**-629.957 ± 0.013**	114.691 ± 0.018	6518.5
CR Volume	-1287.372 ± 0.018	**-959.578 ± 0.012**	327.794 ± 0.021	15421.1

$\mathcal{N}$: 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. However, the numerical results reveal the sense in which model comparison based on these simpler methods can be misleading. There are approximately 115 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 6500 standard deviations of significance[2]. However, there are 327 orders of magnitude more evidence for the log-Normal model for the variability in the cellular retinal volume estimate, with over 15400 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. In any case, 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 CMZ[3], 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 [**?**].

Interestingly, we find that CMZ population and retinal volume estimates are better described by uncorrelated models at all in ages, with approximately 1-8 orders of magnitude of evidence in favour of the uncorrelated model at each age. These data are displayed in **??**. It is interesting to note that the evidence in favour of non-correlation is weakest at 30dpf, which will be discussed below. 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 **??**. 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. 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

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

[3]One 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	**-60.089**	-65.078	4.989
5.0	**-61.349**	-64.873	3.524
8.0	**-60.181**	-67.639	7.458
12.0	**-60.768**	-68.041	7.273
17.0	**-63.503**	-67.344	3.841
23.0	**-69.24**	-76.044	6.804
30.0	**-68.858**	-70.278	1.42
60.0	**-75.769**	-79.67	3.901
90.0	**-74.973**	-83.883	8.91
180.0	**-76.244**	-83.948	7.704
360.0	**-79.822**	-87.628	7.805

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±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 resistent 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 **??**, 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 presented in **??**, 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 [**?**], the onset of this explosive growth seems to come somewhat earlier than the typical metamorphic transition from larval to juvenile stages at 45dpf [**?**]. 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 **??** 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 $\frac{cells}{d}$) is greater than

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 ±95% credible interval for the marginal posterior distribution of the mean.

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 ± 0.04e9 µm^3) completely encompasses the 95% CI on volume at 90dpf (1.11 ± 0.03e9 µm^3), while we assess a 99.6% probability of the 360dpf mean (1.73 ± 0.10e9 µm^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 **??**, 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:

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

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

A model "phase" can then be defined by the CT and $exit$ parameters it applies to update the population and volume over its length. The full parameterisation of a model with p phases is given by p pairs of CT and $exit$ values, $p-1$ phase transition times, and μ_{cv}, which is taken to apply equally to all phases. Given an initial population and volume sampled from the log-Normal models of their interindividual distributions, **??** and **??** 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 [**?**], we calculated the Bayesian evidence for each of 1, 2, and 3-phase models. These results are presented in, with output from maximum a posteriori parameterizations for each model plotted against observations in.

4.4 Characterisation of asymmetrical CMZ population and retinal contribution dynamics

In the course of the preceding investigations, it became apparent that the population asymmetry mentioned in **??** 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 **??**. 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 **??**, 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.

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 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.

As a crude way of assessing the possibility of variable cell cycle attributes across the dorsal-ventral axis, we used the Empirical Bayes approach to estimate the evidence for separate Nowakowski-style [**?**] linear models for the dorsal and ventral CMZ in a cumulative thymidine analogue labelling experiment at 3dpf, against a model for both subpopulations combined. While this model is inadequate for reasons described in **??**, it can serve as a guide as to whether further inferential calculations are likely to be productive here. These results are summarized in **??**, with the relevant linear regressions displayed in **??**.

Table 4.4: **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 values bolded.

The estimated log evidence for the combined model, treating dorsal and ventral CMZ as a single, uniformly proliferative population, is 26.165. Compared to the joint log evidence for two models taking

Figure 4.3: **Developmental progression of dorso-ventral population asymmetry in the CMZ.** Marginal posterior distribution of mean dorsal (D) and ventral (V) population size in 14μm 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.

dorsal and ventral CMZs as separate homogenously proliferative populations, 22.980, there are approximately 3.19 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. 3dpf was selected here for pragmatic reasons; the D/V asymmetry is large and the dorsal population seems to be, perhaps, proliferating more slowly, given its decline relative to the ventral population. It is possible that asymmetric cell cycle times would be better detected during the candidate "time-shift" period, around 15dpf. It is, in any case, plain from the implausibly short calculated cycle times that this model is inadequate.

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 [**?**]. 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-PKCβ 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 **??**, while data from all ages are plotted in **??**. 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 **??**, 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 **??**), 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 assess the joint evidence for Normal and log-Normal models of the data, both separated by age and as age-marginalised measurement sets.

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 [**?**]. 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.

Figure 4.4: **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: PKCβ 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 4.5: **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 14μm coronal cryosections (panel A) or intra-individual D/V count asymmetry ratio (panel B), ±95% credible interval, n=5 animals per age. Data points represent mean counts from three central sections of an experimental animal's eye.

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.

4.7 Toward cell-based computational models of the CMZ

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. Analagous to CMZs in other model organisms, such as X. laevis [**?**], it has been of particular interest to us since the discovery of quiescent stem cells at the mammalian retinal periphery [**?**], 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 [**?**], molecular lesions in a plethora of zebrafish mutants displaying defects in CMZ development and activity remain largely uncharacterised. We examine here one such micropthalmic line identified in an ENU screen, *rys* [**?**], characterised by Wehman et al. as a Class IIA CMZ mutant, with a small eyes (see **??**) 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 [**?**]. 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 [**?**]. 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 [**?**] and localised, in a cell-cycle dependent manner, to histone locus bodies [**?**], induces S-phase entry [**?**] and activates replication-dependent histone gene transcription by direct interaction with histone gene clusters [**?**] in association with histone nuclear factor P (HiNF-P) [**?**]. The protein's effects on S-phase entry and histone transcription are associated with distinct domains at the C-terminus and N-terminus, respectively [**?**]. NPAT is also known to associate

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

with the histone acetyltransferase CBP/p300 [?] and directs histone acetylation by this enzyme [?].

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

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 [?]. 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 disorganised 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 [**?**], 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 **??**. 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 ages[1].

Figure 5.2: ***rys* CMZ populations start relatively small and quiescent, end abberantly large and proliferative**
Panel A: Counts and estimated mean ±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±3.4% of sib RPCs are labelled during the

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

24 hr pulse at 4dpf, while only 24.1±37.2% of *rys* RPCs are. The situation is broadly reversed at 10dpf, with only 24.4±14.9% of sib RPCs labelled, compared to 65.8±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 **??**. 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: ***rys* CMZ RPCs fail to contribute to the neural retina**
M 14µm 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 **??**), 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 postive 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 **??**. 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 mitot-

ically active, this leaves the question of why this apparently proliferative niche's population is declining by 10dpf. We suspected that *rys* 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 *rys* to reliably detect in this manner. In order to investigate this possibility, we assayed the presence of caspase-3 in *rys* and sib CMZs. As displayed in **??**, caspase-3-positive nuclei can be detected in the *rys* CMZ but are not found in sib CMZs, though both display a similar level of central apoptotic activity. The lack of debris observable in *rys* 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 *rys* RPC in the CMZ, displayed in supplementary **??**.

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

Suspecting that *rys* 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 [**?**]. The Pax6-stained region was enlarged in *rys* CMZs relative to their siblings (**??**, center panels). As vsx2 is also known to be a marker of retinal progenitor cells in the CMZ [**?**], we also generated a transgenic vsx2::eGFP *rys* 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 (**??**, 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 20µm coronal cryosection. Middle panels: anti-Pax6 immunohistochemistry. Right panels: GFP expression in a Tg(vsx2:GFP) line introgressed into *rys*.

5.2.2 The micropthalmic 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 **??**. 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 [**?**, **?**], as displayed in **??**.

Figure 5.8: **RT-PCR analysis reveals two aberrant intron retention variants in *rys* 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 *rys* *Danio* npat**

Because *D. rerio* is a teleost known to have undergone genome duplication in an ancestral clade, we used the Synteny Database tool [**?**] identify any possible duplicates; plausibly, a duplication in zebrafish npat could explain the lessened severity of the mutant phenotype when compared to mammalian proviral inactivants. The results of this analysis are presented in supplementary **??**. While npat is in the midst of a region which appears to be duplicated on chromosomes 5 and 15, relative to the unduplicated homologous syntenic 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 **??**, 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 *rys* CMZ RPCs, and with a mutant npat allele reliably linked to the appearance of the *rys* 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 *rys* 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**
20µm 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 [**?**], we hypothesized that the altered nuclear morphology and truncated S-phase we observed in proliferating *rys* 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 *rys* embryo mRNA extracts. These qPCR assays were performed using degenerate primers[2] 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 [**?**], we also set out to determine whether 3' end processing of histone transcripts is altered in *rys*. 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 *rys* cDNA compared to wild type (**??**, 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 *rys*, relative to wild-type. Calculated marginal posterior mean mass over the WT standard, and, for *rys*, the sibling mean, are given in **??**. Even more pronounced in *rys* 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 *rys* 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 *rys* (whether abberantly

[2]Zebrafish 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: ***rys* 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 *rys* 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 *rys* phenotype but to determine if any of its constituent phenomena would appear after an early blockage of *rys* transcription, substantiating the general involvement of npat in *rys*. We tested morpholinos directed both to the npat start codon and to the splice site affected in *rys*, 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 *rys* 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 **??**, 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 *rys* is plausible, as experimental perturbation to npat in WT fish can cause *rys* phenotypic phenomena.

Figure 5.12: **48hpf embryos injected with an npat ATG-targeted morpholino display a small-eye phenotype**
20µm 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. **??**, 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 **??**). 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 abberant 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 [**?**], 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 score[3] 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 [**?**] 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 (presented

[3]That 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 occuring 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 B 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.

in **??**) 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 **??**) 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 likelihood[4]. 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 **??** and **??** respectively, while

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

the inferred combined sources are available in supplementary **??**.

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 *rys* 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 *rys*, 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 progentior markers in these cells. We demonstrate that *rys* RPCs are not, however, arrested in cell cycle. In contradistinction to results finding polyadenylated histone transcript associated with cell cyle arrest [**?**], we find that the mitotic activity of these cells actually accelerates as the overabundance of these transcripts increases. We therefore suggest that the *rys* phenotype is characterised by chromatin disorganisation (phenomenon **??**) and a failure of RPCs to correctly specify (which covers phenomena **??** and **??**) 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 *rys* phenotype, we provide a general, well-evidentiated macromolecular mechanism underlying phenomenon **??** which could plausibly produce phenomena **??** and **??**. That is, the mutant npat's effects on cell-cycle dependent histone transcription and stability may result in abberrant 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 *rys*, together with the gain of novel, abberant 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 *rys*. The observation that the background models explain the positions unique to siblings better than those unique to mutant *rys* 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 [**?**], while others have disavowed the existence of any such code [**?**]. In general, the nucleosome dynamics community has moved on from these discussions in favour of emphasis on rotational sequence preferences [**?**] and translational nucleosome positioning by *trans*-acting factors [**?**]; 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 [**?**]. 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 [**?**] and cell identity and fate [**?**], 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 [**?**]; 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 **??**, 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 invovled 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 specificative behaviours in neural progenitors, which has recently been documented elsewhere in the developing *D. rerio* retina [**?**]. 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 Fruscio1997). The survival and development of *rys* mutant embryos beyond this stage may reflect the presence of wild-type npat transcript contributed maternally [**?**]; *rys* mutants nevertheless do not survive beyond metapmorhosis (~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 **??**. 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 [**?**], 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 [**?**, **?**]. 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 [**?**], 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 [**?**]; 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 [**?**], 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 [**?**, **?**] in pure Julia. It uses the updated Galilean reflection scheme of Skilling's later GMC publication [**?**], 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 [**?**, **?**], 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 ellispoid 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 [**?**], 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_Ensemble` 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_Ensemble` 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_Ensemble`'s `obs`, `priors`, `constants`, and `box` fields, respectively. The `GMC_NS_Ensemble` 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_Ensemble.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_Ensemble)` 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 `convergeensemble!()`. `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 [**?**]. Computational biologists almost immediately benefitted from Skilling's original publication, with the release of the Java-coded nMICA by Down et al. in 2005 [**?**]. 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 guaranteed to converge an ensemble of models on the global optimum likelihood in the parameter space[**?**]. 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 **??** 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 [**?**, **?**] or Galiliean Monte Carlo (GMC) [**?**], as well as methods of increasing computational efficiency, such as dynamic adjustment of ensemble size [**?**].

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 benefit[1].

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 sensical 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 ineffiency 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 **??** until a model more likely than the ensemble contour, given observations, is found, or the instruction's `func_limit` is reached.

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

[1] Probably 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.

9.3 Usage notes

9.4 Recovery of spiked motifs

Part III

Supplementary Materials

Chapter 10

Supplementary materials for ??

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 100μl 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 intesity of 300 lux) in an automated recirculating aquaculture system (Aquaneering). Animals were reared using standard protocols[**?**]. 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. 14μm 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.[**?**]. 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 **??**. 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 **??**, 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 **??**,

??) or PCNA/EdU double-positive (??) 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 14μm 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 ??.

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 ??. Following Nowakowski et al. [?], 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[?]. 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.[?], 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 libary'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 [?]. 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 **??**) 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 [**?**]. 200 iterations were performed. Initially, each iterate involved 250 seeds of each induction timepoint for the count data (panels A-C in Fig **??**) 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 [**?**].

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[**?**] 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 **??** (using the SPSA-optimised parameter sets) and **??** (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 **??** 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 Akiake'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 **??** 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 ??; model output uses the parameterisation given in He et al.[?]. 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 ??; stochastic model output displayed alone.

S3 Fig. SPSA-optimised deterministic mitotic mode model output. As Fig ??; 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 ± 95% CI. Cell cycle length (Tc) and S-phase length (Ts) are estimated from this fit following the method of Nowakowski et al.[**?**]

Chapter 11

Supplementary materials for ??

11.1 Description of the computational cluster used in the work

11.2 Developmental progression of naso-temporal population asymmetry in the CMZ

11.3 Methods

11.3.1 Statistical Analyses

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 14μm transverse cryosections (panel A) or intra-individual N/T count asymmetry ratio (panel B), ±95% credible interval, n=6 animals per age. Data points represent mean counts from three central sections of an experimental animal's eye.

Chapter 12

Supplementary material for ??

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

?? displays the output of the Synteny Database tool [**?**] 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 paralogue 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* npat is highlighted with a red square, visible on the right of the selected Dre15 region, annotated "im:6901964". *H. sapiens* NPAT is highlighted similarly on the left of Hsa11. landmarks of the duplication and rearrangement event that brackets the position of npat on Dre15 are highlighted with red lines.

Figure 12.2: **Rys CMZ RPCs are mitotic at 10dpf**
MIP 14µm 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: Statistical methods and model theory

13.1 Statistical Theory

13.1.1 Bayesian parameter estimation

13.1.1.1 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 [**?**, **?**].

13.1.1.2 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 [**?**], 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.

13.1.2 Bayesian Epistemological View on Model Comparison

13.2 Model Theory

13.2.1 Model optimization

13.2.1.1 MLE

13.2.2 Overfitting, Underfitting

13.2.3 Monte Carlo simulation

Referenced on pages: **??**

13.2.4 The Akiake Information Criterion For Model Selection

13.2.5 Simple Stochastic Models

The Simple Stochastic Model is schematically summarised in Figure **??**. 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

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.

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 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 **??**).

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."[1] [**?**, 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 offsping descended from some particular (usually "initial" and sometimes therefore taken for

[1] While 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.

"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 **??**. 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" [**?**]. 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 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 Siminovitch[2] [**?**], explaining the variability in the size of ectopic spleen colonies formed by hematopoetic stem cells in irradiated mice by means of a Monte Carlo simple stochastic model (SSM). "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." [**?**] 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. [**?**]

[2] It 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.

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 board[3], 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 produces a Gaussian distribution of balls at the bottom of the slope). This metaphor immediately reveals that, *contra* Till et al.'s interpretation of

[3]The 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.

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 begain 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)**[**?**], 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 randomness[1].

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. [**?**]

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 occuring 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

[1] The Kolmogorov complexity of a sequence can be estimated, contrary to common belief[**?**]. 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.

[2] Note 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.
>
> [**?**]

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 [**?**]. 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 [**?**].

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*[**?**].

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 [**?**]. 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"[**?**][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. Moreover, there are good reasons to suppose that the converse is true. In sum, then, we may say that

[3]Bizarrely, 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.

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 process[4]. 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 process[5].

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: attributing one's own ignorance to Nature instead. The current literature of probability

[4]That 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.

[5]Having 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.

> 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." [**?**]

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" [**?**] (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 [**?**], 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 [**?**]. The arrival of new statistics has not helped matters much; serious Bayesian analyses tend to doubt whether medical pharmacological interventions are effective at all [**?**]. 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 [**?**, 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 [**?**].

Moreover, much of the sophisticated literature on mathematical models of stem cell variability are based on interpretations of dynamical systems theory or chaos theory [**?**, **?**] 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 an 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 [**?**, **?**], many other gaily use technical information-theoretic terms like "noise" without any effort to define or measure it [**?**], still others treat the expression of macromolecules as the endpoints of tissue-mechanical forces [**?**], 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 ramification 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. [**?**, 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 [**?**] and that "systems" explanations consist mainly in the complementation of ordinary molecular practice with sophisticated mathematical models [**?**], while acknowledging the increasing space available for the entry of physical explanations in molecular biology [**?**]. 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

?? 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.[**?**, 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 others[6], 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 intrisically 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.

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

[6]Galileo 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.

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 salutory. 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.*" [**?**, 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 [**?**] 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

```
#include <iostream>
#include <string>

#include <cxxtest/TestSuite.h>
#include "ExecutableSupport.hpp"
#include "Exception.hpp"
#include "PetscTools.hpp"
#include "PetscException.hpp"

#include "BoijeCellCycleModel.hpp"
#include "OffLatticeSimulationPropertyStop.hpp"

#include "AbstractCellBasedTestSuite.hpp"

#include "WildTypeCellMutationState.hpp"
#include "TransitCellProliferativeType.hpp"
#include "DifferentiatedCellProliferativeType.hpp"
#include "BoijeRetinalNeuralFates.hpp"

#include "CellsGenerator.hpp"
#include "HoneycombMeshGenerator.hpp"
#include "NodesOnlyMesh.hpp"
#include "NodeBasedCellPopulation.hpp"
#include "VertexBasedCellPopulation.hpp"

#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
                 ↪ values, pass bools as 0 or 1):\n BoijeSimulator
                 ↪ <directoryString> <filenameString>
                 ↪ <outputModeUnsigned(0=counts,1=events,2=sequence)>
                 ↪ <debugOutputBool> <startSeedUnsigned>
                 ↪ <endSeedUnsigned> <endGenerationUnsigned>
                 ↪ <phase2GenerationUnsigned> <phase3GenerationUnsigned>
                 ↪ <pAtoh7Double(0-1)> <pPtf1aDouble(0-1)>
                 ↪ <pngDouble(0-1)>",
38                 true);
39         exit_code = ExecutableSupport::EXIT_BAD_ARGUMENTS;
40         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
       ↪ mode sequence sampling
48     bool debugOutput;
49     unsigned startSeed, endSeed, endGeneration, phase2Generation,
       ↪ phase3Generation;
50     double pAtoh7, pPtf1a, png; //stochastic model parameters
51
52     //PARSE ARGUMENTS
53     directoryString = argv[1];
54     filenameString = argv[2];
55     outputMode = std::stoi(argv[3]);
56     debugOutput = std::stoul(argv[4]);
57     startSeed = std::stoul(argv[5]);
58     endSeed = std::stoul(argv[6]);
59     endGeneration = std::stoul(argv[7]);
```

```
    phase2Generation = std::stoul(argv[8]);
    phase3Generation = std::stoul(argv[9]);
    pAtoh7 = std::stod(argv[10]);
    pPtf1a = std::stod(argv[11]);
    png = std::stod(argv[12]);

    /************************
     * PARAMETER/ARGUMENT SANITY CHECK
     ************************/
    bool sane = 1;

    if (outputMode != 0 && outputMode != 1 && outputMode != 2)
    {
        ExecutableSupport::PrintError(
                "Bad outputMode (argument 3). Must be 0 (counts) 1 (mitotic events) or 2 (sequence sampling)");
        sane = 0;
    }

    if (endSeed < startSeed)
    {
        ExecutableSupport::PrintError("Bad start & end seeds (arguments, 5, 6). endSeed must not be < startSeed");
        sane = 0;
    }

    if (endGeneration <= 0)
    {
        ExecutableSupport::PrintError("Bad endGeneration (argument 7). endGeneration must be > 0");
        sane = 0;
    }

    if (phase3Generation < phase2Generation)
    {
        ExecutableSupport::PrintError(
                "Bad phase2Generation or phase3Generation (arguments 8, 9). phase3Generation must be > phase2Generation. Both must be >0");
        sane = 0;
    }

    if (pAtoh7 < 0 || pAtoh7 > 1)
```

```
    {
        ExecutableSupport::PrintError("Bad pAtoh7 (argument 10). Must be 0-1");
        sane = 0;
    }

    if (pPtf1a < 0 || pPtf1a > 1)
    {
        ExecutableSupport::PrintError("Bad pPtf1a (argument 11). Must be 0-1");
        sane = 0;
    }

    if (png < 0 || png > 1)
    {
        ExecutableSupport::PrintError("Bad png (argument 12). Must be 0-1");
        sane = 0;
    }

    if (sane == 0)
    {
        ExecutableSupport::PrintError("Exiting with bad arguments. See errors for details");
        exit_code = ExecutableSupport::EXIT_BAD_ARGUMENTS;
        return exit_code;
    }

    /************************
     * SIMULATOR OUTPUT SETUP
     ************************/

//Set up singleton LogFile
    LogFile* p_log = LogFile::Instance();
    p_log->Set(0, directoryString, filenameString);

    ExecutableSupport::Print("Simulator writing file " + filenameString + " to directory " + directoryString);

//Log entry counter
    unsigned entry_number = 1;

//Write appropriate headers to log
```

```
    if (outputMode == 0) *p_log << "Entry\tSeed\tCount\n";
    if (outputMode == 1) *p_log << "Time (hpf)\tSeed\tCellID\tMitotic Mode (0=PP;1=PD;2=DD)\n";
    if (outputMode == 2) *p_log << "Entry\tSeed\tSequence\n";

//Instance RNG
    RandomNumberGenerator* p_RNG = RandomNumberGenerator::Instance();

//Initialise pointers to relevant singleton ProliferativeTypes and Properties
    MAKE_PTR(WildTypeCellMutationState, p_state);
    MAKE_PTR(TransitCellProliferativeType, p_Mitotic);
    MAKE_PTR(DifferentiatedCellProliferativeType, p_PostMitotic);
    MAKE_PTR(RetinalGanglion, p_RGC_fate);
    MAKE_PTR(AmacrineHorizontal, p_AC_HC_fate);
    MAKE_PTR(ReceptorBipolar, p_PR_BC_fate);
    MAKE_PTR(CellLabel, p_label);

    /************************
     * SIMULATOR SETUP & RUN
     ************************/

//iterate through supplied seed range, executing one simulation per seed
    for (unsigned seed = startSeed; seed <= endSeed; seed++)
    {
        if (outputMode == 2) *p_log << entry_number << "\t" << seed << "\t"; //write seed to log - sequence written by cellcyclemodel objects

        //initialise pointer to debugWriter
        ColumnDataWriter* debugWriter;

        //initialise SimulationTime (permits cellcyclemodel setup)
        SimulationTime::Instance()->SetStartTime(0.0);

        //Reseed the RNG with the required seed
        p_RNG->Reseed(seed);

        //Initialise a HeCellCycleModel and set it up with appropriate TiL values
        BoijeCellCycleModel* p_cycle_model = new BoijeCellCycleModel;

        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
                         ↪  + "DEBUG_" + std::to_string(seed), false, 10));
178             p_cycle_model->EnableModelDebugOutput(p_debugWriter);
179             debugWriter = &*p_debugWriter;
180         }
181
182         //Setup lineages' cycle model with appropriate parameters
183         p_cycle_model->SetDimension(2);
184         p_cycle_model->SetPostMitoticType(p_PostMitotic);
185
186         //Setup vector containing lineage founder with the properly set
             ↪  up cell cycle model
187         std::vector<CellPtr> cells;
188         CellPtr p_cell(new Cell(p_state, p_cycle_model));
189         p_cell->SetCellProliferativeType(p_Mitotic);
190         p_cycle_model->SetModelParameters(phase2Generation,
             ↪  phase3Generation, pAtoh7, pPtf1a, png);
191         p_cycle_model->SetSpecifiedTypes(p_RGC_fate, p_AC_HC_fate,
             ↪  p_PR_BC_fate);
192         if (outputMode == 2)
             ↪  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
197         //Generate 1×1 mesh for single-cell colony
198         HoneycombMeshGenerator generator(1, 1);
199         MutableMesh<2, 2>* p_generating_mesh = generator.GetMesh();
200         NodesOnlyMesh<2> mesh;
201         mesh.ConstructNodesWithoutMesh(*p_generating_mesh, 1.5);
202
203         //Setup cell population
204         NodeBasedCellPopulation<2>* cell_population(new
             ↪  NodeBasedCellPopulation<2>(mesh, cells));
205
206         //Setup simulator & run simulation
207         boost::shared_ptr<OffLatticeSimulationPropertyStop<2>>
             ↪  p_simulator(
208                 new
                     ↪  OffLatticeSimulationPropertyStop<2>(*cell_population));
```

```
        p_simulator->SetStopProperty(p_Mitotic); //simulation to stop if no mitotic cells are left
        p_simulator->SetDt(0.25);
        p_simulator->SetEndTime(endGeneration);
        p_simulator->SetOutputDirectory("UnusedSimOutput" + filenameString); //unused output
        p_simulator->Solve();

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

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

        //Reset for next simulation
        SimulationTime::Destroy();
        delete cell_population;
        entry_number++;

        if (debugOutput)
        {
            debugWriter->Close();
        }

    }

    p_RNG->Destroy();
    LogFile::Close();

    return exit_code;
}
;
```

15.1.2 /apps/src/GomesSimulator.cpp

```
#include <iostream>
#include <string>

#include <cxxtest/TestSuite.h>
#include "ExecutableSupport.hpp"
#include "Exception.hpp"
```

```
#include "PetscTools.hpp"
#include "PetscException.hpp"

#include "GomesCellCycleModel.hpp"
#include "OffLatticeSimulationPropertyStop.hpp"

#include "AbstractCellBasedTestSuite.hpp"

#include "WildTypeCellMutationState.hpp"
#include "TransitCellProliferativeType.hpp"
#include "DifferentiatedCellProliferativeType.hpp"
#include "GomesRetinalNeuralFates.hpp"

#include "CellsGenerator.hpp"
#include "HoneycombMeshGenerator.hpp"
#include "NodesOnlyMesh.hpp"
#include "NodeBasedCellPopulation.hpp"
#include "VertexBasedCellPopulation.hpp"

#include "ColumnDataWriter.hpp"

int main(int argc, char *argv[])
{
    ExecutableSupport::StartupWithoutShowingCopyright(&argc, &argv);
    //main() returns code indicating sim run success or failure mode
    int exit_code = ExecutableSupport::EXIT_OK;

    if (argc != 15)
    {
        ExecutableSupport::PrintError(
                "Wrong arguments for simulator.\nUsage (replace<> with values, pass bools as 0 or 1):\n GomesSimulator <directoryString> <filenameString> <outputModeUnsigned(0=counts,1=events,2=sequence)> <debugOutputBool> <startSeedUnsigned> <endSeedUnsigned> <endTimeDoubleHours> <cellCycleNormalMeanDouble> <cellCycleNormalStdDouble> <pPPDouble(0-1)> <pPDDouble(0-1)> <pBCDouble(0-1)> <pACDouble(0-1)> <pMGDouble(0-1)>",
                true);
        exit_code = ExecutableSupport::EXIT_BAD_ARGUMENTS;
        return exit_code;
```

```
    }

    /************************
     * SIMULATOR PARAMETERS
     ************************/
    std::string directoryString, filenameString;
    int outputMode; //0 = counts; 1 = mitotic mode events; 2 = mitotic
     ↪  mode sequence sampling
    bool debugOutput;
    unsigned startSeed, endSeed;
    double endTime;
    double normalMu, normalSigma, pPP, pPD, pBC, pAC, pMG; //stochastic
     ↪  model parameters

    //PARSE ARGUMENTS
    directoryString = argv[1];
    filenameString = argv[2];
    outputMode = std::stoi(argv[3]);
    debugOutput = std::stoul(argv[4]);
    startSeed = std::stoul(argv[5]);
    endSeed = std::stoul(argv[6]);
    endTime = std::stod(argv[7]);
    normalMu = std::stod(argv[8]);
    normalSigma = std::stod(argv[9]);
    pPP = std::stod(argv[10]);
    pPD = std::stod(argv[11]);
    pBC = std::stod(argv[12]);
    pAC = std::stod(argv[13]);
    pMG = std::stod(argv[14]);

    /************************
     * PARAMETER/ARGUMENT SANITY CHECK
     ************************/
    bool sane = 1;

    if (outputMode != 0 && outputMode != 1 && outputMode != 2)
    {
        ExecutableSupport::PrintError(
                "Bad outputMode (argument 3). Must be 0 (counts) 1
                 ↪  (mitotic events) or 2 (sequence sampling)");
        sane = 0;
    }

```

```
    if (endSeed < startSeed)
    {
        ExecutableSupport::PrintError("Bad start & end seeds (arguments, 5, 6). endSeed must not be < startSeed");
        sane = 0;
    }

    if (endTime ≤ 0)
    {
        ExecutableSupport::PrintError("Bad endTime (argument 7). endTime must be > 0");
        sane = 0;
    }

    if (normalMu ≤ 0 || normalSigma ≤ 0)
    {
        ExecutableSupport::PrintError("Bad cell cycle normal mean or std (arguments 8, 9). Must be  >0");
        sane = 0;
    }

    if (pPP + pPD > 1 || pPP > 1 || pPP < 0 || pPD > 1 || pPD < 0)
    {
        ExecutableSupport::PrintError(
                "Bad mitotic mode probabilities (arguments 10, 11). pPP + pPD should be ≥0, ≤1, sum should not exceed 1");
        sane = 0;
    }

    if (pBC + pAC + pMG > 1 || pBC > 1 || pBC < 0 || pAC > 1 || pAC < 0 || pMG > 1 || pMG < 0)
    {
        ExecutableSupport::PrintError(
                "Bad specification probabilities (arguments 12, 13, 14). pBC, pAC, pMG should be ≥0, ≤1, sum should not exceed 1");
        sane = 0;
    }

    if (sane == 0)
    {
        ExecutableSupport::PrintError("Exiting with bad arguments. See errors for details");
```

```
        exit_code = ExecutableSupport::EXIT_BAD_ARGUMENTS;
        return exit_code;
    }

    /************************
     * SIMULATOR OUTPUT SETUP
     ************************/

//Set up singleton LogFile
    LogFile* p_log = LogFile::Instance();
    p_log->Set(0, directoryString, filenameString);

    ExecutableSupport::Print("Simulator writing file " + filenameString +
     ↪  " to directory " + directoryString);

//Log entry counter
    unsigned entry_number = 1;

//Write appropriate headers to log
    if (outputMode == 0) *p_log << "Entry\tSeed\tCount\n";
    if (outputMode == 1) *p_log << "Time (hpf)\tSeed\tCellID\tMitotic
     ↪  Mode (0=PP;1=PD;2=DD)\n";
    if (outputMode == 2) *p_log << "Entry\tSeed\tSequence\n";

//Instance RNG
    RandomNumberGenerator* p_RNG = RandomNumberGenerator::Instance();

//Initialise pointers to relevant singleton ProliferativeTypes and
 ↪  Properties
    MAKE_PTR(WildTypeCellMutationState, p_state);
    MAKE_PTR(TransitCellProliferativeType, p_Mitotic);
    MAKE_PTR(DifferentiatedCellProliferativeType, p_PostMitotic);
    MAKE_PTR(RodPhotoreceptor, p_RPh_fate);
    MAKE_PTR(AmacrineCell, p_AC_fate);
    MAKE_PTR(BipolarCell, p_BC_fate);
    MAKE_PTR(MullerGlia, p_MG_fate);
    MAKE_PTR(CellLabel, p_label);

    /************************
     * SIMULATOR SETUP & RUN
     ************************/

//iterate through supplied seed range, executing one simulation per seed
```

```
    for (unsigned seed = startSeed; seed ≤ endSeed; seed++)
    {
        if (outputMode == 2) *p_log << entry_number << "\t" << seed <<
        ↪  "\t"; //write seed to log - sequence written by
        ↪  cellcyclemodel objects

        //initialise pointer to debugWriter
        ColumnDataWriter* debugWriter;

        //initialise SimulationTime (permits cellcyclemodel setup)
        SimulationTime::Instance()→SetStartTime(0.0);

        //Reseed the RNG with the required seed
        p_RNG→Reseed(seed);

        //Initialise a HeCellCycleModel and set it up with appropriate
        ↪  TiL values
        GomesCellCycleModel* p_cycle_model = new GomesCellCycleModel;

        if (debugOutput)
        {
            //Pass ColumnDataWriter to cell cycle model for debug output
            boost::shared_ptr<ColumnDataWriter> p_debugWriter(
                    new ColumnDataWriter(directoryString, filenameString
                    ↪  + "DEBUG_" + std::to_string(seed), false, 10));
            p_cycle_model→EnableModelDebugOutput(p_debugWriter);
            debugWriter = &*p_debugWriter;
        }

        //Setup lineages' cycle model with appropriate parameters
        p_cycle_model→SetDimension(2);
        p_cycle_model→SetPostMitoticType(p_PostMitotic);

        //Setup vector containing lineage founder with the properly set
        ↪  up cell cycle model
        std::vector<CellPtr> cells;
        CellPtr p_cell(new Cell(p_state, p_cycle_model));
        p_cell→SetCellProliferativeType(p_Mitotic);
        p_cycle_model→SetModelParameters(normalMu, normalSigma, pPP,
        ↪  pPD, pBC, pAC, pMG);
        p_cycle_model→SetModelProperties(p_RPh_fate, p_AC_fate,
        ↪  p_BC_fate, p_MG_fate);
```

```
        if (outputMode == 2) p_cycle_model->EnableSequenceSampler(p_label);
        if (outputMode == 2) p_cell->AddCellProperty(p_label);
        p_cell->InitialiseCellCycleModel();
        cells.push_back(p_cell);

        //Generate 1×1 mesh for single-cell colony
        HoneycombMeshGenerator generator(1, 1);
        MutableMesh<2, 2>* p_generating_mesh = generator.GetMesh();
        NodesOnlyMesh<2> mesh;
        mesh.ConstructNodesWithoutMesh(*p_generating_mesh, 1.5);

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

        //Setup simulator & run simulation
        boost::shared_ptr<OffLatticeSimulationPropertyStop<2>> p_simulator(
                new OffLatticeSimulationPropertyStop<2>(*cell_population));
        p_simulator->SetStopProperty(p_Mitotic); //simulation to stop if no mitotic cells are left
        p_simulator->SetDt(0.25);
        p_simulator->SetEndTime(endTime);
        p_simulator->SetOutputDirectory("UnusedSimOutput" + filenameString); //unused output
        p_simulator->Solve();

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

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

        //Reset for next simulation
        SimulationTime::Destroy();
        delete cell_population;
        entry_number++;

        if (debugOutput)
        {
```

```
            debugWriter->Close();
        }

    }

    p_RNG->Destroy();
    LogFile::Close();

    return exit_code;
}
;
```

15.1.3 /apps/src/HeSimulator.cpp

```
#include <iostream>
#include <string>

#include <cxxtest/TestSuite.h>
#include "ExecutableSupport.hpp"
#include "Exception.hpp"
#include "PetscTools.hpp"
#include "PetscException.hpp"

#include "HeCellCycleModel.hpp"
#include "OffLatticeSimulationPropertyStop.hpp"

#include "AbstractCellBasedTestSuite.hpp"

#include "WildTypeCellMutationState.hpp"
#include "TransitCellProliferativeType.hpp"
#include "DifferentiatedCellProliferativeType.hpp"

#include "CellsGenerator.hpp"
#include "HoneycombMeshGenerator.hpp"
#include "NodesOnlyMesh.hpp"
#include "NodeBasedCellPopulation.hpp"
#include "VertexBasedCellPopulation.hpp"

#include "ColumnDataWriter.hpp"

int main(int argc, char *argv[])
{
```

```
    ExecutableSupport::StartupWithoutShowingCopyright(&argc, &argv);
    //main() returns code indicating sim run success or failure mode
    int exit_code = ExecutableSupport::EXIT_OK;

    if (argc != 22 && argc != 20)
    {
        ExecutableSupport::PrintError(
                "Wrong arguments for simulator.\nUsage (replace<> with values, pass bools as 0 or 1):\nStochastic Mode:\nHeSimulator <directoryString> <filenameString> <outputModeUnsigned(0=counts,1=events,2=sequence)> <deterministicBool=0> <fixtureUnsigned(0=He;1=Wan;2=test)> <founderAth5Mutant?Bool> <debugOutputBool> <startSeedUnsigned> <endSeedUnsigned> <inductionTimeDoubleHours> <earliestLineageStartDoubleHours> <latestLineageStartDoubleHours> <endTimeDoubleHours> <mMitoticModePhase2Double> <mMitoticModePhase3Double> <pPP1Double(0-1)> <pPD1Double(0-1)> <pPP1Double(0-1)> <pPD1Double(0-1)> <pPP1Double(0-1)> <pPD1Double(0-1)>\nDeterministic Mode:\nHeSimulator <directoryString> <filenameString> <outputModeUnsigned(0=counts,1=events,2=sequence)> <deterministicBool=1> <fixtureUnsigned(0=He;1=Wan;2=test)> <founderAth5Mutant?Bool> <debugOutputBool> <startSeedUnsigned> <endSeedUnsigned> <inductionTimeDoubleHours> <earliestLineageStartDoubleHours> <latestLineageStartDoubleHours> <endTimeDoubleHours> <phase1ShapeDouble(>0)> <phase1ScaleDouble(>0)> <phase2ShapeDouble(>0)> <phase2ScaleDouble(>0)> <phaseBoundarySisterShiftWidthDouble>\n",
                true);
        exit_code = ExecutableSupport::EXIT_BAD_ARGUMENTS;
        return exit_code;
    }

    /************************
     * SIMULATOR PARAMETERS
     ************************/
    std::string directoryString, filenameString;
```

```
    int outputMode; //0 = counts; 1 = mitotic mode events; 2 = mitotic mode sequence sampling
    bool deterministicMode, ath5founder, debugOutput;
    unsigned fixture, startSeed, endSeed; //fixture 0 = He2012; 1 = Wan2016
    double inductionTime, earliestLineageStartTime, latestLineageStartTime, endTime;
    double mitoticModePhase2, mitoticModePhase3, pPP1, pPD1, pPP2, pPD2, pPP3, pPD3; //stochastic model parameters
    double phase1Shape, phase1Scale, phase2Shape, phase2Scale, phaseSisterShiftWidth, phaseOffset;

    //PARSE ARGUMENTS
    directoryString = argv[1];
    filenameString = argv[2];
    outputMode = std::stoi(argv[3]);
    deterministicMode = std::stoul(argv[4]);
    fixture = std::stoul(argv[5]);
    ath5founder = std::stoul(argv[6]);
    debugOutput = std::stoul(argv[7]);
    startSeed = std::stoul(argv[8]);
    endSeed = std::stoul(argv[9]);
    inductionTime = std::stod(argv[10]);
    earliestLineageStartTime = std::stod(argv[11]);
    latestLineageStartTime = std::stod(argv[12]);
    endTime = std::stod(argv[13]);

    if (deterministicMode == 0)
    {
        mitoticModePhase2 = std::stod(argv[14]);
        mitoticModePhase3 = std::stod(argv[15]);
        pPP1 = std::stod(argv[16]);
        pPD1 = std::stod(argv[17]);
        pPP2 = std::stod(argv[18]);
        pPD2 = std::stod(argv[19]);
        pPP3 = std::stod(argv[20]);
        pPD3 = std::stod(argv[21]);
    }
    else if (deterministicMode == 1)
    {
        phase1Shape = std::stod(argv[14]);
        phase1Scale = std::stod(argv[15]);
        phase2Shape = std::stod(argv[16]);
```

```
        phase2Scale = std::stod(argv[17]);
        phaseSisterShiftWidth = std::stod(argv[18]);
        phaseOffset = std::stod(argv[19]);
    }
    else
    {
        ExecutableSupport::PrintError("Bad deterministicMode (argument 4). Must be 0 or 1");
        exit_code = ExecutableSupport::EXIT_BAD_ARGUMENTS;
        return exit_code;
    }

    /************************
     * PARAMETER/ARGUMENT SANITY CHECK
     ************************/
    bool sane = 1;

    if (outputMode != 0 && outputMode != 1 && outputMode != 2)
    {
        ExecutableSupport::PrintError(
                "Bad outputMode (argument 3). Must be 0 (counts) 1 (mitotic events) or 2 (sequence sampling)");
        sane = 0;
    }

    if (fixture != 0 && fixture != 1 && fixture != 2)
    {
        ExecutableSupport::PrintError("Bad fixture (argument 5). Must be 0 (He), 1 (Wan), or 2 (validation/test)");
        sane = 0;
    }

    if (ath5founder != 0 && ath5founder != 1)
    {
        ExecutableSupport::PrintError("Bad ath5founder (argument 6). Must be 0 (wild type) or 1 (ath5 mutant)");
        sane = 0;
    }

    if (endSeed < startSeed)
    {
        ExecutableSupport::PrintError("Bad start & end seeds (arguments, 8, 9). endSeed must not be < startSeed");
```

```
        sane = 0;
    }

    if (inductionTime ≥ endTime)
    {
        ExecutableSupport::PrintError("Bad latestLineageStartTime (argument 10). Must be <endTime(arg13)");
        sane = 0;
    }
    if (earliestLineageStartTime ≥ endTime || earliestLineageStartTime ≥ latestLineageStartTime)
    {
        ExecutableSupport::PrintError(
                "Bad earliestLineageStartTime (argument 11). Must be <endTime(arg13), <latestLineageStarTime (arg12)");
        sane = 0;
    }
    if (latestLineageStartTime > endTime)
    {
        ExecutableSupport::PrintError("Bad latestLineageStartTime (argument 12). Must be ≤endTime(arg13)");
        sane = 0;
    }

    if (deterministicMode == 0)
    {
        if (mitoticModePhase2 < 0)
        {
            ExecutableSupport::PrintError("Bad mitoticModePhase2 (argument 14). Must be >0");
            sane = 0;
        }
        if (mitoticModePhase3 < 0)
        {
            ExecutableSupport::PrintError("Bad mitoticModePhase3 (argument 15). Must be >0");
            sane = 0;
        }
        if (pPP1 + pPD1 > 1 || pPP1 > 1 || pPP1 < 0 || pPD1 > 1 || pPD1 < 0)
        {
            ExecutableSupport::PrintError(
```

```
                    "Bad phase 1 probabilities (arguments 16, 17). pPP1 + pPD1 should be ⩾0, ⩽1, sum should not exceed 1");
            sane = 0;
        }
        if (pPP2 + pPD2 > 1 || pPP2 > 1 || pPP2 < 0 || pPD2 > 1 || pPD2 < 0)
        {
            ExecutableSupport::PrintError(
                    "Bad phase 2 probabilities (arguments 18, 19). pPP2 + pPD2 should be ⩾0, ⩽1, sum should not exceed 1");
            sane = 0;
        }
        if (pPP3 + pPD3 > 1 || pPP3 > 1 || pPP3 < 0 || pPD3 > 1 || pPD3 < 0)
        {
            ExecutableSupport::PrintError(
                    "Bad phase 3 probabilities (arguments 20, 21). pPP3 + pPD3 should be ⩾0, ⩽1, sum should not exceed 1");
            sane = 0;
        }
    }

    if (deterministicMode == 1)
    {
        if (phase1Shape ⩽ 0)
        {
            ExecutableSupport::PrintError("Bad phase1Shape (argument 14). Must be >0");
            sane = 0;
        }
        if (phase1Scale ⩽ 0)
        {
            ExecutableSupport::PrintError("Bad phase1Scale (argument 15). Must be >0");
            sane = 0;
        }
        if (phase2Shape ⩽ 0)
        {
            ExecutableSupport::PrintError("Bad phase1Shape (argument 16). Must be >0");
```

```
            sane = 0;
        }
        if (phase2Scale ≤ 0)
        {
            ExecutableSupport::PrintError("Bad phase1Scale (argument 17). Must be >0");
            sane = 0;
        }
        if (phaseSisterShiftWidth ≤ 0)
        {
            ExecutableSupport::PrintError("Bad phaseSisterShiftWidth (argument 18). Must be >0");
            sane = 0;
        }
    }

    if (sane == 0)
    {
        ExecutableSupport::PrintError("Exiting with bad arguments. See errors for details");
        exit_code = ExecutableSupport::EXIT_BAD_ARGUMENTS;
        return exit_code;

    }

    /************************
     * SIMULATOR OUTPUT SETUP
     ************************/

//Set up singleton LogFile
    LogFile* p_log = LogFile::Instance();
    p_log->Set(0, directoryString, filenameString);

    ExecutableSupport::Print("Simulator writing file " + filenameString + " to directory " + directoryString);

//Log entry counter
    unsigned entry_number = 1;

//Write appropriate headers to log
    if (outputMode == 0) *p_log << "Entry\tInduction Time (h)\tSeed\tCount\n";
```

```
    if (outputMode == 1) *p_log << "Time (hpf)\tSeed\tCellID\tMitotic Mode (0=PP;1=PD;2=DD)\n";
    if (outputMode == 2) *p_log << "Entry\tSeed\tSequence\n";

//Instance RNG
    RandomNumberGenerator* p_RNG = RandomNumberGenerator::Instance();

//Initialise pointers to relevant singleton ProliferativeTypes and Properties
    MAKE_PTR(WildTypeCellMutationState, p_state);
    MAKE_PTR(TransitCellProliferativeType, p_Mitotic);
    MAKE_PTR(DifferentiatedCellProliferativeType, p_PostMitotic);
    MAKE_PTR(Ath5Mo, p_Morpholino);
    MAKE_PTR(CellLabel, p_label);

    /************************
     * SIMULATOR SETUP & RUN
     ************************/

//iterate through supplied seed range, executing one simulation per seed
    for (unsigned seed = startSeed; seed <= endSeed; seed++)
    {
        if (outputMode == 2) *p_log << entry_number << "\t" << seed << "\t"; //write seed to log - sequence written by cellcyclemodel objects

        //initialise pointer to debugWriter
        ColumnDataWriter* debugWriter;

        //initialise SimulationTime (permits cellcyclemodel setup)
        SimulationTime::Instance()->SetStartTime(0.0);

        //Reseed the RNG with the required seed
        p_RNG->Reseed(seed);

        //Initialise a HeCellCycleModel and set it up with appropriate TiL values
        HeCellCycleModel* p_cycle_model = new HeCellCycleModel;

        if (debugOutput)
        {
            //Pass ColumnDataWriter to cell cycle model for debug output
            boost::shared_ptr<ColumnDataWriter> p_debugWriter(
```

```
                    new ColumnDataWriter(directoryString, filenameString + "DEBUG_" + std::to_string(seed), false, 10));
            p_cycle_model->EnableModelDebugOutput(p_debugWriter);
            debugWriter = &*p_debugWriter;
        }

        double currTiL; //Time in Lineage offset for lineages induced after first mitosis
        double lineageStartTime; //first mitosis time (hpf)
        double currSimEndTime; //simulation end time (hpf);


        /*************************************************************************
         * Time in Lineage Generation Fixtures & Cell Cycle Model Setup
*************************************************************************/

        if (fixture == 0) //He 2012-type fixture - even distribution across nasal-temporal axis
        {
            //generate lineage start time from even random distro across earliest-latest start time figures
            lineageStartTime = (p_RNG->ranf() * (latestLineageStartTime - earliestLineageStartTime))
                    + earliestLineageStartTime;
            //this reflects induction of cells after the lineages' first mitosis
            if (lineageStartTime < inductionTime)
            {
                currTiL = inductionTime - lineageStartTime;
                currSimEndTime = endTime - inductionTime;
                if (outputMode == 1) p_cycle_model->EnableModeEventOutput(inductionTime, seed);
            }
            //if the lineage starts after the induction time, give it zero & TiL run the appropriate-length simulation
            //(ie. the endTime is reduced by the amount of time after induction that the first mitosis occurs)
            if (lineageStartTime ≥ inductionTime)
            {
                currTiL = 0.0;
                currSimEndTime = endTime - lineageStartTime;
```

```
                if (outputMode == 1) p_cycle_model->EnableModeEventOutput(lineageStartTime, seed);
            }

        }
        else if (fixture == 1) //Wan 2016-type fixture - each lineage founder selected randomly across residency time, simulator allowed to run until end of residency time
        //passing residency time (as latestLineageStartTime) and endTime separately allows for creation of "shadow CMZ" population
        //this allows investigation of different assumptions about how Wan et al.'s model output was generated
        {
            //generate random lineage start time from even random distro across CMZ residency time
            currTiL = p_RNG->ranf() * latestLineageStartTime;
            currSimEndTime = std::max(.05, endTime - currTiL); //minimum 1 timestep, prevents 0 timestep SimulationTime error
            if (outputMode == 1) p_cycle_model->EnableModeEventOutput(0, seed);
        }
        else if (fixture == 2) //validation fixture- all founders have TiL given by induction time
        {
            currTiL = inductionTime;
            currSimEndTime = endTime;
        }

        //Setup lineages' cycle model with appropriate parameters
        p_cycle_model->SetDimension(2);
        //p_cycle_model->SetPostMitoticType(p_PostMitotic);

        if (!deterministicMode)
        {
            p_cycle_model->SetModelParameters(currTiL, mitoticModePhase2, mitoticModePhase2 + mitoticModePhase3, pPP1,
                                              pPD1, pPP2, pPD2, pPP3, pPD3);
        }
        else
        {
            //Gamma-distribute phase3 boundary
```

```
            double currPhase2Boundary = phaseOffset + p_RNG->GammaRandomDeviate(phase1Shape, phase1Scale);
            double currPhase3Boundary = currPhase2Boundary + p_RNG->GammaRandomDeviate(phase2Shape, phase2Scale);

            p_cycle_model->SetDeterministicMode(currTiL, currPhase2Boundary, currPhase3Boundary, phaseSisterShiftWidth);
        }

        if (outputMode == 2) p_cycle_model->EnableSequenceSampler();

        //Setup vector containing lineage founder with the properly set up cell cycle model
        std::vector<CellPtr> cells;
        CellPtr p_cell(new Cell(p_state, p_cycle_model));
        p_cell->SetCellProliferativeType(p_Mitotic);
        if (ath5founder == 1) p_cell->AddCellProperty(p_Morpholino);
        if (outputMode == 2) p_cell->AddCellProperty(p_label);
        p_cell->InitialiseCellCycleModel();
        cells.push_back(p_cell);

        //Generate 1×1 mesh for single-cell colony
        HoneycombMeshGenerator generator(1, 1);
        MutableMesh<2, 2>* p_generating_mesh = generator.GetMesh();
        NodesOnlyMesh<2> mesh;
        mesh.ConstructNodesWithoutMesh(*p_generating_mesh, 1.5);

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

        //Setup simulator & run simulation
        boost::shared_ptr<OffLatticeSimulationPropertyStop<2>> p_simulator(
                new OffLatticeSimulationPropertyStop<2>(*cell_population));
        p_simulator->SetStopProperty(p_Mitotic); //simulation to stop if no mitotic cells are left
        p_simulator->SetDt(0.05);
        p_simulator->SetEndTime(currSimEndTime);
        p_simulator->SetOutputDirectory("UnusedSimOutput" + filenameString); //unused output
```

```
        p_simulator->Solve();

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

        if (outputMode == 0) *p_log << entry_number << "\t" <<
        ↪  inductionTime << "\t" << seed << "\t" << count << "\n";
        if (outputMode == 2) *p_log << "\n";

        //Reset for next simulation
        SimulationTime::Destroy();
        delete cell_population;
        entry_number++;

        if (debugOutput)
        {
            debugWriter->Close();
        }

    }

    p_RNG->Destroy();
    LogFile::Close();

    return exit_code;
}
;
```

15.1.4 /apps/src/WanSimDebug.cpp

```
#include <iostream>
#include <string>

#include <cxxtest/TestSuite.h>
#include "ExecutableSupport.hpp"
#include "Exception.hpp"
#include "PetscTools.hpp"
#include "PetscException.hpp"

#include "WanStemCellCycleModel.hpp"
#include "HeCellCycleModel.hpp"
#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 +
       ↪  " to directory " + directoryString);
54
```

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

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

    //Setup cell population
    boost::shared_ptr<NodeBasedCellPopulation<2>> cell_population(new
    ↪   NodeBasedCellPopulation<2>(mesh, cells));

    ↪   cell_population->AddCellPopulationCountWriter<CellProliferativeTypesCountWrit

    p_stem_model->EnableExpandingStemPopulation(1, cell_population);

    //Setup simulator & run simulation
    boost::shared_ptr<OffLatticeSimulation<2>> p_simulator(
            new OffLatticeSimulation<2>(*cell_population));
    p_simulator->SetDt(1);
    p_simulator->SetOutputDirectory(directoryString + "/WanDebug");
    p_simulator->SetEndTime(100);
    p_simulator->Solve();

    std::vector<unsigned> prolTypes =
    ↪   cell_population->GetCellProliferativeTypeCount();
    unsigned realCells = cell_population->GetNumRealCells();
    unsigned allCells = cell_population->GetNumAllCells();

    Timer::Print("stem: " + std::to_string(prolTypes[0]) + " transit: " +
    ↪   std::to_string(prolTypes[1]) + " postmitotic: " +
    ↪   std::to_string(prolTypes[2]));
    Timer::Print("realcells: " + std::to_string(realCells) + " allcells:
    ↪   " + std::to_string(allCells));

    //Reset for next simulation
    SimulationTime::Destroy();
    cell_population.reset();

    p_RNG->Destroy();
    LogFile::Close();

    return exit_code;
}
;
```

15.1.5 /apps/src/WanSimulator.cpp

```
#include <iostream>
#include <string>

#include <cxxtest/TestSuite.h>
#include "ExecutableSupport.hpp"
#include "Exception.hpp"
#include "PetscTools.hpp"
#include "PetscException.hpp"

#include "WanStemCellCycleModel.hpp"
#include "HeCellCycleModel.hpp"
#include "OffLatticeSimulationPropertyStop.hpp"

#include "AbstractCellBasedTestSuite.hpp"

#include "WildTypeCellMutationState.hpp"
#include "StemCellProliferativeType.hpp"
#include "TransitCellProliferativeType.hpp"
#include "DifferentiatedCellProliferativeType.hpp"

#include "CellsGenerator.hpp"
#include "HoneycombMeshGenerator.hpp"
#include "NodesOnlyMesh.hpp"
#include "NodeBasedCellPopulation.hpp"
#include "VertexBasedCellPopulation.hpp"

#include "CellProliferativeTypesCountWriter.hpp"

int main(int argc, char *argv[])
{
    ExecutableSupport::StartupWithoutShowingCopyright(&argc, &argv);
    //main() returns code indicating sim run success or failure mode
    int exit_code = ExecutableSupport::EXIT_OK;

    if (argc != 23)
    {
        ExecutableSupport::PrintError(
```

```
                "Wrong arguments for simulator.\nUsage (replace<> with values, pass bools as 0 or 1):\n WanSimulator <directoryString> <startSeedUnsigned> <endSeedUnsigned> <cmzResidencyTimeDoubleHours> <stemDivisorDouble> <meanProgenitorPopualtion@3dpfDouble> <stdProgenitorPopulation@3dpfDouble> <stemGammaShiftDouble> <stemGammaShapeDouble> <stemGammaScaleDouble> <progenitorGammaShiftDouble> <progenitorGammaShapeDouble> <progenitorGammaScaleDouble> <progenitorSisterShiftDouble> <mMitoticModePhase2Double> <mMitoticModePhase3Double> <pPP1Double(0-1)> <pPD1Double(0-1)> <pPP1Double(0-1)> <pPD1Double(0-1)> <pPP1Double(0-1)> <pPD1Double(0-1)>",
                true);
        exit_code = ExecutableSupport::EXIT_BAD_ARGUMENTS;
        return exit_code;
    }

    /***********************
     * SIMULATOR PARAMETERS
     ***********************/
    std::string directoryString;
    unsigned startSeed, endSeed;
    double cmzResidencyTime;
    double stemDivisor, progenitorMean, progenitorStd;
    double stemGammaShift, stemGammaShape, stemGammaScale, progenitorGammaShift, progenitorGammaShape,
            progenitorGammaScale, progenitorGammaSister;
    double mitoticModePhase2, mitoticModePhase3, pPP1, pPD1, pPP2, pPD2, pPP3, pPD3; //stochastic He model parameters

    //PARSE ARGUMENTS
    directoryString = argv[1];
    startSeed = std::stoul(argv[2]);
    endSeed = std::stoul(argv[3]);
    cmzResidencyTime = std::stod(argv[4]);
    //starting cell number distributions
    stemDivisor = std::stod(argv[5]);
    progenitorMean = std::stod(argv[6]);
    progenitorStd = std::stod(argv[7]);
```

```
    //cycle duration params
    stemGammaShift = std::stod(argv[8]);
    stemGammaShape = std::stod(argv[9]);
    stemGammaScale = std::stod(argv[10]);
    progenitorGammaShift = std::stod(argv[11]);
    progenitorGammaShape = std::stod(argv[12]);
    progenitorGammaScale = std::stod(argv[13]);
    progenitorGammaSister = std::stod(argv[14]);
    //He model params
    mitoticModePhase2 = std::stod(argv[15]);
    mitoticModePhase3 = std::stod(argv[16]);
    pPP1 = std::stod(argv[17]);
    pPD1 = std::stod(argv[18]);
    pPP2 = std::stod(argv[19]);
    pPD2 = std::stod(argv[20]);
    pPP3 = std::stod(argv[21]);
    pPD3 = std::stod(argv[22]);

    std::vector<double> stemOffspringParams = { mitoticModePhase2, mitoticModePhase2 + mitoticModePhase3, pPP1, pPD1,
                                                pPP2, pPD2, pPP3, pPD3, progenitorGammaShift, progenitorGammaShape,
                                                progenitorGammaScale, progenitorGammaSister };

    /************************
     * PARAMETER/ARGUMENT SANITY CHECK
     ************************/
    bool sane = 1;

    if (endSeed < startSeed)
    {
        ExecutableSupport::PrintError("Bad start & end seeds (arguments, 3, 4). endSeed must not be < startSeed");
        sane = 0;
    }

    if (cmzResidencyTime ≤ 0)
    {
        ExecutableSupport::PrintError("Bad CMZ residency time (argument 5). cmzResidencyTime must be positive-valued");
```

```
        sane = 0;
    }

    if (stemDivisor ≤ 0)
    {
        ExecutableSupport::PrintError("Bad stemDivisor (argument 6). stemDivisor must be positive-valued");
        sane = 0;
    }

    if (progenitorMean ≤ 0 || progenitorStd ≤ 0)
    {
        ExecutableSupport::PrintError("Bad progenitorMean or progenitorStd (arguments 7,8). Must be positive-valued");
        sane = 0;
    }

    if (stemGammaShift < 0 || stemGammaShape ≤ 0 || stemGammaScale ≤ 0)
    {
        ExecutableSupport::PrintError(
                "Bad stemGammaShift, stemGammaShape, or stemGammaScale (arguments 9, 10, 11). Shifts must be ≥0, cycle shape and scale params must be positive-valued");
        sane = 0;
    }

    if (progenitorGammaShift < 0 || progenitorGammaShape ≤ 0 || progenitorGammaScale ≤ 0 || progenitorGammaSister < 0)
    {
        ExecutableSupport::PrintError(
                "Bad progenitorGammaShift, progenitorGammaShape, progenitorGammaScale, or progenitorGammaSisterShift (arguments 12,13,14,15). Shifts must be ≥0, cycle shape and scale params must be positive-valued");
        sane = 0;
    }

    if (mitoticModePhase2 < 0)
    {
        ExecutableSupport::PrintError("Bad mitoticModePhase2 (argument 14). Must be >0");
        sane = 0;
    }
```

```
    if (mitoticModePhase3 < 0)
    {
        ExecutableSupport::PrintError("Bad mitoticModePhase3 (argument 15). Must be >0");
        sane = 0;
    }
    if (pPP1 + pPD1 > 1 || pPP1 > 1 || pPP1 < 0 || pPD1 > 1 || pPD1 < 0)
    {
        ExecutableSupport::PrintError(
                "Bad phase 1 probabilities (arguments 16, 17). pPP1 + pPD1 should be ⩾0, ⩽1, sum should not exceed 1");
        sane = 0;
    }
    if (pPP2 + pPD2 > 1 || pPP2 > 1 || pPP2 < 0 || pPD2 > 1 || pPD2 < 0)
    {
        ExecutableSupport::PrintError(
                "Bad phase 2 probabilities (arguments 18, 19). pPP2 + pPD2 should be ⩾0, ⩽1, sum should not exceed 1");
        sane = 0;
    }
    if (pPP3 + pPD3 > 1 || pPP3 > 1 || pPP3 < 0 || pPD3 > 1 || pPD3 < 0)
    {
        ExecutableSupport::PrintError(
                "Bad phase 3 probabilities (arguments 20, 21). pPP3 + pPD3 should be ⩾0, ⩽1, sum should not exceed 1");
        sane = 0;
    }

    if (sane == 0)
    {
        ExecutableSupport::PrintError("Exiting with bad arguments. See errors for details");
        exit_code = ExecutableSupport::EXIT_BAD_ARGUMENTS;
        return exit_code;
    }

    /************************
     * SIMULATOR SETUP & RUN
     ************************/

    ExecutableSupport::Print("Simulator writing files to directory " + directoryString);

```

```
//Instance RNG
    RandomNumberGenerator* p_RNG = RandomNumberGenerator::Instance();

//Initialise pointers to relevant singleton ProliferativeTypes and Properties
    boost::shared_ptr<AbstractCellProperty> p_state(CellPropertyRegistry::Instance()->Get<WildTypeCellMutationState>());
    boost::shared_ptr<AbstractCellProperty> p_Stem(CellPropertyRegistry::Instance()->Get<StemCellProliferativeType>());
    boost::shared_ptr<AbstractCellProperty> p_Transit(CellPropertyRegistry::Instance()->Get<TransitCellProliferativeType>()
    boost::shared_ptr<AbstractCellProperty> p_PostMitotic(CellPropertyRegistry::Instance()->Get<DifferentiatedCellProliferative

//iterate through supplied seed range, executing one simulation per seed
    for (unsigned seed = startSeed; seed <= endSeed; seed++)
    {
        //initialise SimulationTime (permits cellcyclemodel setup)
        SimulationTime::Instance()->SetStartTime(0.0);

        //Reseed the RNG with the required seed
        p_RNG->Reseed(seed);

        //unsigned numberStem = int(std::round(p_RNG->NormalRandomDeviate(stemMean, stemStd)));
        unsigned numberProgenitors = int(std::round(p_RNG->NormalRandomDeviate(progenitorMean, progenitorStd)));
        unsigned numberStem = int(std::round(numberProgenitors / stemDivisor));

        std::vector<CellPtr> stems;
        std::vector<CellPtr> cells;

        for (unsigned i = 0; i < numberStem; i++)
        {
            WanStemCellCycleModel* p_stem_model = new WanStemCellCycleModel;
            p_stem_model->SetDimension(2);
```

```
            p_stem_model->SetModelParameters(stemGammaShift, stemGammaShape, stemGammaScale, stemOffspringParams);

            CellPtr p_cell(new Cell(p_state, p_stem_model));
            p_cell->InitialiseCellCycleModel();
            stems.push_back(p_cell);
            cells.push_back(p_cell);
        }

        for (unsigned i = 0; i < numberProgenitors; i++)
        {
            double currTiL = p_RNG->ranf() * cmzResidencyTime;

            HeCellCycleModel* p_prog_model = new HeCellCycleModel;
            p_prog_model->SetDimension(2);
            p_prog_model->SetModelParameters(currTiL, mitoticModePhase2, mitoticModePhase2 + mitoticModePhase3, pPP1,
                                             pPD1, pPP2, pPD2, pPP3, pPD3);
            p_prog_model->EnableKillSpecified();

            CellPtr p_cell(new Cell(p_state, p_prog_model));
            p_cell->InitialiseCellCycleModel();
            cells.push_back(p_cell);
        }

        //Generate 1x#cells mesh for abstract colony
        HoneycombMeshGenerator generator(1, (numberProgenitors + numberStem));
        MutableMesh<2, 2>* p_generating_mesh = generator.GetMesh();
        NodesOnlyMesh<2> mesh;
        mesh.ConstructNodesWithoutMesh(*p_generating_mesh, 1.5);

        //Setup cell population
        boost::shared_ptr<NodeBasedCellPopulation<2>> cell_population(new NodeBasedCellPopulation<2>(mesh, cells));

        cell_population->AddCellPopulationCountWriter<CellProliferativeTypesCount

        //Give Wan stem cells the population & base stem pop size
        for (auto p_cell : stems)
        {
```

```
            WanStemCellCycleModel* p_cycle_model = dynamic_cast<WanStemCellCycleModel*>(p_cell->GetCellCycleModel());
            p_cycle_model->EnableExpandingStemPopulation(numberStem, cell_population);
        }

        //Setup simulator & run simulation
        boost::shared_ptr<OffLatticeSimulationPropertyStop<2>> p_simulator(
                new OffLatticeSimulationPropertyStop<2>(*cell_population));
        p_simulator->SetStopProperty(p_Transit); //simulation to stop if no RPCs are left
        p_simulator->SetDt(1);
        p_simulator->SetOutputDirectory(directoryString + "/Seed" + std::to_string(seed) + "Results");
        p_simulator->SetEndTime(8568); // 360dpf - 3dpf simulation start time
        p_simulator->Solve();

        //Reset for next simulation
        SimulationTime::Destroy();
        cell_population.reset();
    }

    p_RNG->Destroy();

    return exit_code;
}
;
```

15.1.6 /python_fixtures/He_output_fixture.py

```
import multiprocessing
import os
import subprocess
import datetime

import numpy as np
from imageio.plugins._bsdf import BsdfSerializer

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
```

```

#STOCHASTIC MITOTIC MODE
#HE ORIGINAL PARAMETERS
#mitotic mode per-phase probabilities
#3-phase mitotic mode time periodisation
o_mitotic_mode_phase_2 = 8 #These are phase lengths, so
o_mitotic_mode_phase_3 = 7 #Phase3 boundary = mmp2 + mmp3
o_phase_1_pPP = 1.0
o_phase_1_pPD = 0.0
o_phase_2_pPP = 0.2
o_phase_2_pPD = 0.4
o_phase_3_pPP = 0.2
o_phase_3_pPD = 0.0

#SPSA REFIT PARAMETERS
#mitotic mode per-phase probabilities
#3-phase mitotic mode time periodisation
mitotic_mode_phase_2 = 4.1483 #These are phase lengths, so
mitotic_mode_phase_3 = 11.6416 #Phase3 boundary = mmp2 + mmp3
phase_1_pPP = 1.0
phase_1_pPD = 0.0
phase_2_pPP = 0.1959
phase_2_pPD = 0.5168
phase_3_pPP = 0.2934
phase_3_pPD = 0.0
he_model_params = 15

#DETERMINISTIC MITOTIC MODE
#Phase boundary shift parameters
phase_1_shape = 3.7371
phase_1_scale = 1.8114
phase_2_shape = 2.5769
phase_2_scale = 1.6814
phase_sister_shift_widths = 1.6326
phase_offset = 1.2333
det_model_params = 12

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)
```

```
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)
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)

def main():

    command_list = []

    for m in range(0,len(run_modes)):
        curr_list = []

        run_mode_string = str(run_modes[m])
        deterministic_mode = run_modes[m]
        if run_modes[m] == 2:
            deterministic_mode = 0
        command_count = 0 #for iterating seed numbers
        base_command = executable

        #induction count commands
        for i in range(0,len(induction_times)):

            output_mode = 0 #0=lineage counts;1=mitotic event logging;2=sequence sampling
            fixture = 0 #0=He 2012;1=Wan 2016
            curr_start_seed = start_seed + command_count * (end_seed_counts+1)
            curr_end_seed = curr_start_seed + end_seed_counts
            ath5founder = 0

            command = base_command+" "\
                        +count_directory+" "\
                        +count_filenames[0]+str(induction_times[i])+"SDMode"+run_
                        "\
                        +str(output_mode)+" "\
                        +str(deterministic_mode)+" "\
                        +str(fixture)+" "\
                        +str(ath5founder)+" "\
```

```
                        +str(debug_output)+" "\
                        +str(curr_start_seed)+" "\
                        +str(curr_end_seed)+" "\
                        +str(induction_times[i])+" "\
                        +str(earliest_lineage_start_time)+" "\
                        +str(latest_lineage_start_time)+" "\
                        +str(counts_end_time)+" "
            curr_list.append(command)
            command_count += 1

        #wan command
        output_mode = 0
        fixture = 1
        curr_start_seed = start_seed + command_count * (end_seed_counts+1)
        curr_end_seed = curr_start_seed + end_seed_counts
        ath5founder = 0
        wan_command = base_command+" "\
                        +count_directory+" "\
                        +count_filenames[1]+"SDMode"+run_mode_string+" "\
                        +str(output_mode)+" "\
                        +str(deterministic_mode)+" "\
                        +str(fixture)+" "\
                        +str(ath5founder)+" "\
                        +str(debug_output)+" "\
                        +str(curr_start_seed)+" "\
                        +str(curr_end_seed)+" "\
                        +str(0)+" "\
                        +str(0)+" "\
                        +str(wan_residency_time)+" "\
                        +str(counts_end_time)+" "
        curr_list.append(wan_command)
        command_count += 1

        #wan- 17 hr constraint- no "shadow RPCs"
        curr_start_seed = start_seed + command_count * (end_seed_counts+1)
        curr_end_seed = curr_start_seed + end_seed_counts
        wan_command = base_command+" "\
                        +count_directory+" "\

                        +count_filenames[1]+"NoShadSDMode"+run_mode_string+" "\
```

```
                        +str(output_mode)+" "\
                        +str(deterministic_mode)+" "\
                        +str(fixture)+" "\
                        +str(ath5founder)+" "\
                        +str(debug_output)+" "\
                        +str(curr_start_seed)+" "\
                        +str(curr_end_seed)+" "\
                        +str(0)+" "\
                        +str(0)+" "\
                        +str(wan_residency_time)+" "\
                        +str(wan_residency_time)+" "
        curr_list.append(wan_command)
        command_count += 1

        #ath5 command
        output_mode = 0
        fixture = 2
        curr_start_seed = start_seed + command_count *
         ↪  (end_seed_counts+1)
        curr_end_seed = curr_start_seed + end_seed_counts
        ath5founder = 1

        ath5_command = base_command+" "\
                        +count_directory+" "\
                        +count_filenames[2]+"SDMode"+run_mode_string+" "\
                        +str(output_mode)+" "\
                        +str(deterministic_mode)+" "\
                        +str(fixture)+" "\
                        +str(ath5founder)+" "\
                        +str(debug_output)+" "\
                        +str(curr_start_seed)+" "\
                        +str(curr_end_seed)+" "\
                        +str(0)+" "\
                        +str(earliest_lineage_start_time)+" "\
                        +str(latest_lineage_start_time)+" "\
                        +str(counts_end_time)+" "
        curr_list.append(ath5_command)
        command_count += 1

        #validate counts command
        curr_start_seed = start_seed + command_count *
         ↪  (end_seed_counts+1)
        curr_end_seed = curr_start_seed + end_seed_counts
```

```
        ath5founder = 0

        validate_command = base_command+" "\
                        +count_directory+" "\
                        +count_filenames[3]+"SDMode"+run_mode_string+" "\
                        +str(output_mode)+" "\
                        +str(deterministic_mode)+" "\
                        +str(fixture)+" "\
                        +str(ath5founder)+" "\
                        +str(debug_output)+" "\
                        +str(curr_start_seed)+" "\
                        +str(curr_end_seed)+" "\
                        +str(0)+" "\
                        +str(earliest_lineage_start_time)+" "\
                        +str(latest_lineage_start_time)+" "\
                        +str(counts_end_time)+" "
        curr_list.append(validate_command)
        command_count += 1

        #mitotic mode rate command
        output_mode = 1
        fixture = 0
        curr_start_seed = start_seed + command_count *
         ↪  (end_seed_counts+1)
        curr_end_seed = curr_start_seed + end_seed_events
        mode_rate_command = base_command+" "\
                        +event_directory+" "\
                        +event_filenames[0]+"SDMode"+run_mode_string+" "\
                        +str(output_mode)+" "\
                        +str(deterministic_mode)+" "\
                        +str(fixture)+" "\
                        +str(ath5founder)+" "\
                        +str(debug_output)+" "\
                        +str(curr_start_seed)+" "\
                        +str(curr_end_seed)+" "\
                        +str(earliest_lineage_start_time)+" "\
                        +str(earliest_lineage_start_time)+" "\
                        +str(latest_lineage_start_time)+" "\
                        +str(events_end_time)+" "

        curr_list.append(mode_rate_command)
        command_count += 1

```

```
        #validate rate command
        fixture = 2
        curr_start_seed = start_seed + command_count *
         ↪  (end_seed_counts+1)
        curr_end_seed = curr_start_seed + end_seed_events
        validate_rate_command = base_command+" "\
                        +event_directory+" "\
                        +event_filenames[1]+"SDMode"+run_mode_string+" "\
                        +str(output_mode)+" "\
                        +str(deterministic_mode)+" "\
                        +str(fixture)+" "\
                        +str(ath5founder)+" "\
                        +str(debug_output)+" "\
                        +str(curr_start_seed)+" "\
                        +str(curr_end_seed)+" "\
                        +str(0)+" "\
                        +str(earliest_lineage_start_time)+" "\
                        +str(latest_lineage_start_time)+" "\
                        +str(events_end_time)+" "

        curr_list.append(validate_rate_command)
        command_count += 1

        for i in range(0,len(curr_list)):
            if run_modes[m] == 2:
                curr_list[i] = curr_list[i] + original_theta_string
            if run_modes[m] == 0:
                curr_list[i] = curr_list[i] + stochastic_theta_string
            if run_modes[m] == 1:
                curr_list[i] = curr_list[i] + deterministic_theta_string

        command_list = command_list + curr_list

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

    print(command_list)

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

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

```

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


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


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

15.1.7 /python_fixtures/Kolmogorov_fixture.py

```
import multiprocessing
import os
import subprocess
import datetime
import numpy as np
from imageio.plugins._bsdf import BsdfSerializer

gomes_executable = '/home/main/chaste_build/projects/ISP/apps/GomesSimulator'
he_executable = '/home/main/chaste_build/projects/ISP/apps/HeSimulator'
boije_executable = '/home/main/chaste_build/projects/ISP/apps/BoijeSimulator'

empirical_data = '/home/main/git/chaste/projects/ISP/empirical_data/empirical_lineages.csv'

if not(os.path.isfile(gomes_executable)):
    raise Exception('Could not find executable: ' + gomes_executable)
if not(os.path.isfile(he_executable)):
    raise Exception('Could not find executable: ' + he_executable)
if not(os.path.isfile(boije_executable)):
    raise Exception('Could not find executable: ' + boije_executable)

#########################
# SIMULATION PARAMETERS
```

```
##########################

#Define start and end RNG seeds; determines:
#No. lineages per loss function run
#unique sequence of RNG results for each lineage

directory = "KolmogorovSequences"
empirical_sequences_name = "EOsequences"
output_mode = 2 #sequence sampler
debug_output = 0 #0=off;1=on
start_seed = 0
end_seed = 9999

##########################
#GLOBAL MODEL PARAMETERS
##########################

number_traversal_lineages = 60

############################
# SPECIFIC MODEL PARAMETERS
############################

#GOMES MODEL
g_end_time = 480
g_normal_mean = 3.9716
g_normal_sigma = .32839
g_pPP = .055
g_pPD = .221
g_pBC = .128
g_pAC = .106
g_pMG = .028

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)

#HE MODEL - GENERAL
h_fixture = 2
h_ath5founder = 0
```

```
h_start_time = 0
h_end_time = 80

#HE MODEL - STOCHASTIC
h_deterministic_mode = 0
h_mitotic_mode_phase_2 = 8 #These are phase lengths, so
h_mitotic_mode_phase_3 = 7 #Phase3 boundary = mmp2 + mmp3
h_phase_1_pPP = 1.0
h_phase_1_pPD = 0.0
h_phase_2_pPP = 0.2
h_phase_2_pPD = 0.4
h_phase_3_pPP = 0.2
h_phase_3_pPD = 0.0

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)

#HE MODEL -STOCHASTIC (REFIT)
hr_mitotic_mode_phase_2 = 4.1483 #These are phase lengths, so
hr_mitotic_mode_phase_3 = 11.6416 #Phase3 boundary = mmp2 + mmp3
hr_phase_1_pPP = 1.0
hr_phase_1_pPD = 0.0
hr_phase_2_pPP = 0.1959
hr_phase_2_pPD = 0.5168
hr_phase_3_pPP = 0.2934
hr_phase_3_pPD = 0.0
```

```
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)

#HE MODEL - DETERMINISTIC ALTERNATIVE
d_deterministic_mode = 1
#Phase boundary shift parameters
d_phase_1_shape = 3.7371
d_phase_1_scale = 1.8114
d_phase_2_shape = 2.5769
d_phase_2_scale = 1.6814
d_phase_sister_shift_widths = 1.6326
d_phase_offset = 1.2333

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)

#BOIJE MODEL
b_end_generation = 250
b_phase_2_generation = 3
b_phase_3_generation = 5
b_pAtoh7 = .32
b_pPtf1a = .3
b_png = .8
```

```
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)

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

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

def main():

    command_list = []
    command_list.append(g_string)
    command_list.append(h_string)
    command_list.append(hr_string)
    command_list.append(d_string)
    command_list.append(b_string)

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

    print(command_list)

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

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

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

    traverse_lineages(empirical_data)

def traverse_lineages(data_filename):

    mode_sequences = [None]*(end_seed+1)
```

```

    #load lineage tracing data
    e_data = np.loadtxt(data_filename, skiprows = 1, usecols = (0,1,2,3))

    #reproducible RNG
    p_RNG = np.random.RandomState(seed = 0)

    k=0

    for curr_seed in range (start_seed,end_seed+1):
        random_lineage_number = p_RNG.randint(1,number_traversal_lineages)
        lineage_events = np.array(e_data[np.where(e_data[:,0]==random_lineage_number)])

        #find mitotic mode of first event and write to log
        current_event = 1
        mitotic_mode = int(lineage_events[np.where(lineage_events[:,1]==current_event)][:,3])
        mode_sequences[k] = f'{mitotic_mode:.0f}'

        while mitotic_mode != 2:
            if mitotic_mode == 1 and p_RNG.random_sample() < .5: break

            child_events = lineage_events[np.where(lineage_events[:,2]==current_event)]
            random_child_row = p_RNG.randint(0,np.ma.size(child_events,0))
            current_event= child_events[random_child_row,1]
            mitotic_mode = child_events[random_child_row,3]
            mode_sequences[k] = mode_sequences[k] + f'{mitotic_mode:.0f}'

        e_file.write(str(k+1)+"\t"+mode_sequences[k]+"\n")
        k += 1

    return mode_sequences

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

```

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

15.1.8 /python_fixtures/SPSA_fixture.py

```
import multiprocessing
import os
import subprocess
import datetime

import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import bernoulli
from fileinput import filename
from statsmodels.sandbox.distributions.gof_new import a_st70_upp

executable = '/home/main/chaste_build/projects/ISP/apps/HeSimulator'

if not(os.path.isfile(executable)):
    raise Exception('Could not find executable: ' + executable)

#####################
# SPSA COEFFICIENTS
#####################

a_s = .025
c_s = .1

a_d = .0019
c_d = .1

A = 20 #10% of expected # iterations
alpha = .602
gamma = .101

###########################
# SPSA & AIC UTILITY VARS
###########################

max_iterations = 199
```

```
number_comparisons = 3030 #total number of comparison points between model output and empirical data (1000 per induction time, 10 per rate mode)
number_comparisons_per_induction = 1000
error_samples = 5000 #number of samples to draw when estimating plausibility interval for simulations

#########################
# SIMULATION PARAMETERS
#########################

#Define start and end RNG seeds; determines:
#No. lineages per loss function run
#unique sequence of RNG results for each lineage
start_seed = 0
end_seed = 249
rate_end_seed = 99
directory_name = "SPSA"
file_name_he = "HeSPSA"
file_name_det = "DetSPSA"
log_name = "HeSPSAOutput"
deterministic_modes = [0, 1] #1=det. mode enabled
count_output_mode = 0 #0=lineage counts;1=mitotic event logging;2=sequence sampling
event_output_mode = 1
fixture = 0 #0=He 2012;1=Wan 2016
debug_output = 0 #0=off;1=on
ath5founder = 0 #0=no morpholino 1=ath5 morpholino

##########################
#GLOBAL MODEL PARAMETERS
##########################

#Values defining different marker induction timepoints & relative start time of TiL counter
earliest_lineage_start_time = 23.0 #RPCs begin to enter "He model regime" nasally at 23hpf
latest_lineage_start_time = 39.0 #last temporal retinal RPC has entered "He model regime" at 39hpf
induction_times = [ 24, 32, 48 ]
end_time = 72.0
rate_end_time = 80

```

```
########################################
# SPECIFIC MODEL PARAMETERS - THETAHAT
########################################

#STOCHASTIC MITOTIC MODE
#mitotic mode per-phase probabilities
#3-phase mitotic mode time periodisation
mitotic_mode_phase_2 = 8 #These are phase lengths, so
mitotic_mode_phase_3 = 7 #Phase3 boundary = mmp2 + mmp3
phase_1_pPP = 1.0
phase_1_pPD = 0.0
phase_2_pPP = 0.2
phase_2_pPD = 0.4
phase_3_pPP = 0.2
phase_3_pPD = 0.0
he_model_params = 15

#DETERMINISTIC MITOTIC MODE
#Phase boundary shift parameters
phase_1_shape = 3
phase_1_scale = 2
phase_2_shape = 2
phase_2_scale = 2
phase_sister_shift_widths = .25
phase_offset = 0
det_model_params = 13

#array "theta_spsa" is manipulated during SPSA calculations, these are starting point references
stochastic_theta_zero = np.array([mitotic_mode_phase_2, mitotic_mode_phase_3, phase_2_pPP, phase_2_pPD, phase_3_pPP ])
determinsitic_theta_zero = np.array([phase_1_shape, phase_1_scale, phase_2_shape, phase_2_scale, phase_sister_shift_widths, phase_offset])

#scales ak gain sequence for probability variables
prob_scale_vector = np.array([ 1, 1, .1, .1, .1])
shift_scale_vector = np.array([1, 1, 1, 1, 1, 3])

##############################
# HE ET AL EMPIRICAL RESULTS
##############################

```

```
count_bin_sequence = np.arange(1,32,1)
count_x_sequence = np.arange(1,31,1)
count_trim_value = 30

rate_bin_sequence = np.arange(30,85,5)
rate_x_sequence = np.arange(30,80,5)
rate_trim_value = 10

#Count probability arrays
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
raw_counts_24 = raw_counts[0:64,:]
raw_counts_32 = raw_counts[64:233,:]
raw_counts_48 = raw_counts[233:396,:]

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
prob_32,bin_edges = np.histogram(np.sum(raw_counts_32,axis=1),count_bin_sequence,density=True)
prob_48,bin_edges = np.histogram(np.sum(raw_counts_48,axis=1),count_bin_sequence,density=True)

#extend histogram to 1000 - allows large lineages of some iterates to be included in AIC comparison
prob_empirical_24 = np.concatenate([prob_24, np.zeros(int(number_comparisons_per_induction) - prob_24.size)])
prob_empirical_32 = np.concatenate([prob_32, np.zeros(int(number_comparisons_per_induction) - prob_32.size)])
prob_empirical_48 = np.concatenate([prob_48, np.zeros(int(number_comparisons_per_induction) - prob_48.size)])

count_prob_list = [prob_empirical_24, prob_empirical_32, prob_empirical_48]

#no. of lineages observed per induction timepoint/event group
lineages_sampled_24 = 64
lineages_sampled_32 = 169
lineages_sampled_48 = 163
lineages_sampled_events = 60
```

```
lineages_sampled_list = [lineages_sampled_24,lineages_sampled_32,lineages_sampled_48]

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

observed_PP = observed_events[np.where(observed_events[:,0]==0)]
observed_PD = observed_events[np.where(observed_events[:,0]==1)]
observed_DD = observed_events[np.where(observed_events[:,0]==2)]

histo_PP,bin_edges = np.histogram(observed_PP,rate_bin_sequence,density=False)
histo_PD,bin_edges = np.histogram(observed_PD,rate_bin_sequence,density=False)
histo_DD,bin_edges = np.histogram(observed_DD,rate_bin_sequence,density=False)

#hourly per-lineage probabilities- NOT probability density function
prob_empirical_PP = np.array((histo_PP/lineages_sampled_events)/5)
prob_empirical_PD = np.array((histo_PD/lineages_sampled_events)/5)
prob_empirical_DD = np.array((histo_DD/lineages_sampled_events)/5)

event_prob_list = [prob_empirical_PP,prob_empirical_PD,prob_empirical_DD]

#setup the log file, appending to any existing results
log_filename = "/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/" +directory_name +"/" + log_name
os.makedirs(os.path.dirname(log_filename), exist_ok=True)

log = open(log_filename,"w")

#plotting utility stuff
#interactive plot mode
plt.ion()
#setup new iterate plot
fig, ((plt0, plt1, plt2),(plt3, plt4, plt5)) = plt.subplots(2,3,figsize=(12,6))
plot_list = [plt0,plt1,plt2,plt3,plt4,plt5]
```

```python


def main():
    global theta_spsa, a,c,file_name,scale_vector,end_seed,rate_end_seed,alpha,gamma

    for m in range(0,len(deterministic_modes)):

        #traceable RNG
        p_RNG = np.random.RandomState(seed=786)

        deterministic_mode = deterministic_modes[m]
        if deterministic_mode == 0:
            theta_spsa = stochastic_theta_zero
            a = a_s
            c = c_s
            file_name = file_name_he
            scale_vector = prob_scale_vector
            now = datetime.datetime.now()
            log.write("Began SPSA optimisation of He model @\n" + str(datetime.datetime.now()) + "\n")
            log.write("k\tphase2\tphase3\tPP2\tPD2\tPP3\n")
            number_params = he_model_params
        if deterministic_mode == 1:
            theta_spsa = determinsitic_theta_zero
            a = a_d
            c = c_d
            file_name = file_name_det
            scale_vector = shift_scale_vector
            now = datetime.datetime.now()
            log.write("Began SPSA optimisation of deterministic model @\n" + str(datetime.datetime.now()) + "\n")
            log.write("k\tp1Sh\tp1Sc\tp2Sh\tp2Sc\tsisterShift\toffset\n")
            number_params = det_model_params

        #SPSA algorithm iterator k starts at 0
        k=0

        while k ≤ max_iterations:
            if k == 0:
                end_seed = 249
                rate_end_seed = 99

```

```
            #@defined iterate, increase # of seed to decrease RNG noise to low level
            if k == 169:
                end_seed = 999
                rate_end_seed = 249


            #@defined iterate, increase # of seeds to decrease RNG noise to close to nil, switch to asymptotically optimal alpha and gamma
            if k == 189:
                end_seed = 4999
                rate_end_seed = 1249
                alpha = 1
                gamma = (1/6)

            #write the parameter set to be evaluated to file
            if deterministic_mode == 0: log.write(str(k) + "\t" + str(theta_spsa[0]) + "\t" + str(theta_spsa[1]) + "\t" + str(theta_spsa[2]) + "\t" + str(theta_spsa[3]) + "\t" + str(theta_spsa[4]) + "\n")
            if deterministic_mode == 1: log.write(str(k) + "\t" + str(theta_spsa[0]) + "\t" + str(theta_spsa[1]) + "\t" + str(theta_spsa[2]) + "\t" + str(theta_spsa[3]) + "\t" + str(theta_spsa[4])+ "\t" + str(theta_spsa[5]) + "\n")

            #populate the deltak perturbation vector with samples from a .5p bernoulli +1 -1 distribution
            delta_k = bernoulli.rvs(.5, size=theta_spsa.size, random_state=p_RNG)
            delta_k[delta_k == 0] = -1

            ak = a / ((A + k + 1)**alpha) #calculate ak from gain sequence
            scaled_ak = ak * scale_vector #scale ak appropriately for parameters expressed in hrs & percent

            ck = c / ((k + 1)**gamma) #calculate ck from gain sequence
            scaled_ck = ck * scale_vector

            #Project theta_spsa into space bounded by ck to allow gradient sampling at bounds of probability space
```

```
            projected_theta = project_theta(theta_spsa, scaled_ck, deterministic_mode)

            #Calculate theta+ and theta- vectors for gradient estimate
            theta_plus = projected_theta + scaled_ck * delta_k
            theta_minus = projected_theta - scaled_ck * delta_k

            AIC_gradient = evaluate_AIC_gradient(k, theta_plus, theta_minus, deterministic_mode, number_params, file_name)

            ghat = ((AIC_gradient) / (2 * ck)) * delta_k

            log.write("ghat0: " + str(ghat[0]) + "\n")

            #update new theta_spsa
            theta_spsa = theta_spsa - scaled_ak * ghat

            #constrain updated theta_spsa
            theta_spsa = project_theta(theta_spsa, np.zeros(theta_spsa.size), deterministic_mode)

            k+=1

        #write final result and close log
        if deterministic_mode == 0: log.write(str(k) + "\t" + str(theta_spsa[0]) + "\t" + str(theta_spsa[1]) + "\t" + str(theta_spsa[2]) + "\t" + str(theta_spsa[3]) + "\t" + str(theta_spsa[4]) + "\n")
        if deterministic_mode == 1: log.write(str(k) + "\t" + str(theta_spsa[0]) + "\t" + str(theta_spsa[1]) + "\t" + str(theta_spsa[2]) + "\t" + str(theta_spsa[3]) + "\t" + str(theta_spsa[4])+ "\t" + str(theta_spsa[5]) + "\n")

    log.close

def project_theta(theta, boundary, deterministic_mode):
# Required projection is onto unit triangle modified by boundary (ck value)
# Values outside the lower bounds are reset first
# Then, for phase 2 probabilities, we project (PPa,PDa)→(PPb,PDb) such that if( (PPa+ck) + (PDa+ck) >1), PPb+ck + PDb +ck = 1.
# This takes care of the upper bound
```

```
    #project all negative parameters back into bounded space
    if deterministic_mode == False:
        for i in range(0, theta.size):
            if i == 0:
                if theta[i] < boundary[i] + 4.0: theta[i] = boundary[i] + 4.0 #project mitotic_mode_phase_2 to a minimum of 4- below this param has no effect (due to refractory period after first division)
            else:
                if theta[i] < boundary[i]: theta[i] = boundary[i]

        #if PP3 exceeds 1-boundary, project back to 1-boundary
        if theta[4] > 1 - boundary[4]: theta[4] = 1 - boundary[4]

        if theta[2] + theta[3] > 1 - boundary[2]: #if pPP2 + pPD2 gives a total beyond the current boundary-reduced total of 1.0

            if theta[2] > (1 - 2 * boundary[2]) and theta[2] ≤ theta[3] - (1 - 2 * boundary[2]): # these (PP,PD) points will project into negative PD space
                theta[2] = 1 - 2 * boundary[2]
                theta[3] = boundary[2]

            elif theta[3] > (1 - 2 * boundary[2]) and theta[2] ≤ theta[3] - (1 - 2 * boundary[2]): # these (PP,PD) points will project into negative PP space
                theta[3] = 1 - 2 * boundary[2]
                theta[2] = boundary[2]

            else: # the (PP,PD) points can be projected onto the line PD = -PP + 1 and modified by the boundary;
                v1 = [ 1, -1 ] # vector from (0,1) to (1,0)
                v2 = [ theta[2], theta[3] - 1 ] #vector from (0,1) to (PP,PD)
                dot_product = np.dot(v1,v2)
                lengthv1 = 2
                theta[2] = (dot_product / lengthv1) - boundary[2]
                theta[3] = (1 - dot_product / lengthv1) - boundary[2]

    if deterministic_mode == True:
        for i in range(0, theta.size-1): #exclude offset param
```

```
            if theta[i] < boundary[i]: theta[i] = boundary[i] + 0.1 #prevents non→0 arguments for deterministic mode

        #projects sister shift value such that 95% of sister shift values will be less than smallest mean phase time
        minPhase = min(theta[0]*theta[1],theta[2]*theta[3])
        if theta[4] > minPhase/2 - boundary[4]: theta[4] = minPhase/2 - boundary[4]

    return theta

def evaluate_AIC_gradient(k, theta_plus, theta_minus, deterministic_mode, number_params, file_name):
    #Form the simulator commands for current thetas and deterministic modes
    command_list = []
    base_command = executable

    rate_settings = str(event_output_mode)+" "\
                    +str(deterministic_mode)+" "\
                    +str(fixture)+" "\
                    +str(ath5founder)+" "\
                    +str(debug_output)+" "\
                    +str(start_seed)+" "\
                    +str(rate_end_seed)+" "\
                    +str(earliest_lineage_start_time)+" "\
                    +str(earliest_lineage_start_time)+" "\
                    +str(latest_lineage_start_time)+" "\
                    +str(rate_end_time)+" "

    count_settings_1 = str(count_output_mode)+" "\
                       +str(deterministic_mode)+" "\
                       +str(fixture)+" "\
                       +str(ath5founder)+" "\
                       +str(debug_output)+" "\
                       +str(start_seed)+" "\
                       +str(end_seed)+" "

    count_settings_2 = str(earliest_lineage_start_time)+" "\
                       +str(latest_lineage_start_time)+" "\
                       +str(end_time)+" "

    if deterministic_mode == 0:
```

```
        stochastic_params_plus = str(theta_plus[0])+" "\
                    +str(theta_plus[1])+" "\
                    +str(phase_1_pPP)+" "\
                    +str(phase_1_pPD)+" "\
                    +str(theta_plus[2])+" "\
                    +str(theta_plus[3])+" "\
                    +str(theta_plus[4])+" "\
                    +str(phase_3_pPD)

        stochastic_params_minus = str(theta_minus[0])+" "\
                    +str(theta_minus[1])+" "\
                    +str(phase_1_pPP)+" "\
                    +str(phase_1_pPD)+" "\
                    +str(theta_minus[2])+" "\
                    +str(theta_minus[3])+" "\
                    +str(theta_minus[4])+" "\
                    +str(phase_3_pPD)

        command_rate_plus = base_command\
                    +" "+directory_name+" "+ file_name+"RatePlus "\
                    +rate_settings\
                    +stochastic_params_plus

        command_rate_minus = base_command\
                    +" "+directory_name+" "+ file_name+"RateMinus "\
                    +rate_settings\
                    +stochastic_params_minus

        command_list.append(command_rate_plus)
        command_list.append(command_rate_minus)

        for i in range(0,len(induction_times)):
            command_plus = base_command\
                        +" "+directory_name+"
                        ↪  "+file_name+str(induction_times[i])+"Plus "\
                        +count_settings_1\
                        +str(induction_times[i])+" "\
                        +count_settings_2\
                        +stochastic_params_plus

            command_minus = base_command\
```

```
                        +" "+directory_name+"
                         ↪  "+file_name+str(induction_times[i])+"Minus "\
                        +count_settings_1\
                        +str(induction_times[i])+" "\
                        +count_settings_2\
                        +stochastic_params_minus

            command_list.append(command_plus)
            command_list.append(command_minus)

    if deterministic_mode == 1:

        deterministic_params_plus = str(theta_plus[0])+" "\
                    +str(theta_plus[1])+" "\
                    +str(theta_plus[2])+" "\
                    +str(theta_plus[3])+" "\
                    +str(theta_plus[4])+" "\
                    +str(theta_plus[5])

        deterministic_params_minus = str(theta_minus[0])+" "\
                    +str(theta_minus[1])+" "\
                    +str(theta_minus[2])+" "\
                    +str(theta_minus[3])+" "\
                    +str(theta_minus[4])+" "\
                    +str(theta_minus[5])

        command_rate_plus = base_command\
                    +" "+directory_name+" "+ file_name+"RatePlus "\
                    +rate_settings\
                    +deterministic_params_plus

        command_rate_minus = base_command\
                    +" "+directory_name+" "+ file_name+"RateMinus "\
                    +rate_settings\
                    +deterministic_params_minus

        command_list.append(command_rate_plus)
        command_list.append(command_rate_minus)

        for i in range(0,len(induction_times)):
            command_plus = base_command\
                        +" "+directory_name+"
                         ↪  "+file_name+str(induction_times[i])+"Plus "\
```

```
                        +count_settings_1\
                        +str(induction_times[i])+" "\
                        +count_settings_2\
                        +deterministic_params_plus

            command_minus = base_command\
                        +" "+directory_name+"
                         ↪  "+file_name+str(induction_times[i])+"Minus "\
                        +count_settings_1\
                        +str(induction_times[i])+" "\
                        +count_settings_2\
                        +deterministic_params_minus

            command_list.append(command_plus)
            command_list.append(command_minus)


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

    print("Starting simulations for iterate " + str(k) + " with " +
     ↪  str(cpu_count) + " processes, " + str(end_seed+1) + " lineages
     ↪  simulated for counts, " + str(rate_end_seed+1) + " lineages for
     ↪  events")

    log.flush() #required to prevent pool from jamming up log for some
     ↪  reason

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

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

    #these numpy arrays hold the individual RSS vals for timepoints +
     ↪  rates
    rss_plus = np.zeros(len(induction_times)+3)
    rss_minus = np.zeros(len(induction_times)+3)

    #clear the plots on the interactive figure
    for i in range(0, len(plot_list)):
        plt.sca(plot_list[i])
```

```
        plt.cla()

    #calculate RSS for each induction timepoint
    for i in range(0,len(induction_times)):
        counts_plus = np.loadtxt("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/" + directory_name + "/" + file_name + str(induction_times[i]) + "Plus", skiprows=1, usecols=3)
        counts_minus = np.loadtxt("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/" + directory_name + "/" + file_name + str(induction_times[i]) + "Minus", skiprows=1, usecols=3)
        prob_histo_plus, bin_edges = np.histogram(counts_plus, bins=1000, range=(1,1001),density=True)
        prob_histo_minus, bin_edges = np.histogram(counts_minus, bins=1000, range=(1,1001),density=True)

        residual_plus = prob_histo_plus - count_prob_list[i]
        residual_minus = prob_histo_minus - count_prob_list[i]

        rss_plus[i] = np.sum(np.square(residual_plus))
        rss_minus[i] = np.sum(np.square(residual_minus))

        plotter(plot_list[i], counts_plus, counts_minus, count_prob_list[i],lineages_sampled_list[i], 0)


    rates_plus = np.loadtxt("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/" + directory_name + "/" + file_name + "RatePlus", skiprows=1, usecols=(0,3))
    rates_minus = np.loadtxt("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/" + directory_name + "/" + file_name + "RateMinus", skiprows=1, usecols=(0,3))

    for i in range (0,3):
        mode_rate_plus = np.array(rates_plus[np.where(rates_plus[:,1]==i)])
        mode_rate_minus = np.array(rates_minus[np.where(rates_minus[:,1]==i)])

```

```
        histo_rate_plus, bin_edges = np.histogram(mode_rate_plus[:,0], rate_bin_sequence, density=False)
        histo_rate_minus, bin_edges = np.histogram(mode_rate_minus[:,0], rate_bin_sequence, density=False)

        #hourly per-lineage probabilities
        prob_histo_rate_plus = np.array(histo_rate_plus / ((rate_end_seed + 1)*5))
        prob_histo_rate_minus = np.array(histo_rate_plus / ((rate_end_seed + 1)*5))

        residual_plus = prob_histo_rate_plus - event_prob_list[i]
        residual_minus = prob_histo_rate_minus - event_prob_list[i]

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

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

        plotter(plot_list[i+3], mode_rate_plus[:,0], mode_rate_minus[:,0], event_prob_list[i],lineages_sampled_events, 1)

    AIC_plus = 2 * number_params + number_comparisons * np.log(np.sum(rss_plus))
    AIC_minus = 2 * number_params + number_comparisons * np.log(np.sum(rss_minus))

    plt.show()

    plt.savefig("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/"+directory_name +"/" + "SDmode" + str(deterministic_mode) + "iterate" + str(k) + ".png")

    log.write("theta_plus:\n")
    if deterministic_mode == 0: log.write(str(k) + "\t" + str(theta_plus[0]) + "\t" + str(theta_plus[1]) + "\t" + str(theta_plus[2]) + "\t" + str(theta_plus[3]) + "\t" + str(theta_plus[4]) + "\n")
```

```
    if deterministic_mode == 1: log.write(str(k) + "\t" + str(theta_plus[0]) + "\t" + str(theta_plus[1]) + "\t" + str(theta_plus[2]) + "\t" + str(theta_plus[3]) + "\t" + str(theta_plus[4]) + "\t" + str(theta_plus[5]) +"\n")
    log.write("theta_minus:\n")
    if deterministic_mode == 0: log.write(str(k) + "\t" + str(theta_minus[0]) + "\t" + str(theta_minus[1]) + "\t" + str(theta_minus[2]) + "\t" + str(theta_minus[3]) + "\t" + str(theta_minus[4]) + "\n")
    if deterministic_mode == 1: log.write(str(k) + "\t" + str(theta_minus[0]) + "\t" + str(theta_minus[1]) + "\t" + str(theta_minus[2]) + "\t" + str(theta_minus[3]) + "\t" + str(theta_minus[4]) + "\t" + str(theta_minus[5]) + "\n")

    log.write("PositiveAIC: " + str(AIC_plus) + " NegativeAIC: " + str(AIC_minus) + "\n")

    AIC_gradient_sample = AIC_plus - AIC_minus;

    return AIC_gradient_sample

#data plotter function for monitoring SPSA results
def plotter(subplot,plus,minus,empirical_prob,samples,mode):
    global plt0,plt1,plt2,plt3,plt4,plt5, prob_histo_plus, prob_histo_minus
    bin_sequence = []
    x_sequence = []
    trim_value = 0

    if mode == 0:
        bin_sequence = count_bin_sequence
        x_sequence = count_x_sequence
        trim_value = count_trim_value
        prob_histo_plus, bin_edges = np.histogram(plus, bin_sequence, density=True)
        prob_histo_minus, bin_edges = np.histogram(minus, bin_sequence, density=True)

    if mode == 1:
        bin_sequence = rate_bin_sequence
        x_sequence = rate_x_sequence
        trim_value = rate_trim_value
```

```
        histo_plus, bin_edges = np.histogram(plus, bin_sequence, density=False)
        histo_minus, bin_edges = np.histogram(minus, bin_sequence, density=False)
        prob_histo_plus = np.array(histo_plus / ((rate_end_seed + 1)*5))
        prob_histo_minus = np.array(histo_plus / ((rate_end_seed + 1)*5))

    trimmed_prob = empirical_prob[0:trim_value]

    subplot.plot(x_sequence,trimmed_prob, 'k+')
    plt.pause(0.0001)

    interval_plus = sampler(plus,samples,bin_sequence)

    subplot.plot(x_sequence,prob_histo_plus, 'g-')
    plt.pause(0.0001)
    subplot.fill_between(x_sequence, (prob_histo_plus - interval_plus), (prob_histo_plus + interval_plus), alpha=0.2, edgecolor='#008000', facecolor='#00FF00')
    plt.pause(0.0001)

    interval_minus = sampler(minus,samples,bin_sequence)

    subplot.plot(x_sequence,prob_histo_minus, 'm-')
    plt.pause(0.0001)
    subplot.fill_between(x_sequence, (prob_histo_minus - interval_minus), (prob_histo_minus + interval_minus), alpha=0.2, edgecolor='#800080', facecolor='#FF00FF')
    plt.pause(0.0001)

    if subplot==plt0: plt0.set_ylim((0,.20))
    if subplot==plt1: plt1.set_ylim(0,.20)
    if subplot==plt2: plt2.set_ylim(0,.7)
    if subplot==plt3: plt3.set_ylim(0, .30)
    if subplot==plt4: plt4.set_ylim(0, .15)
    if subplot==plt5: plt5.set_ylim(0, .35)

def sampler(data,samples,bin_sequence):

    if len(data) == 0: #catches edge case w/ no entries for a mitotic mode
        data=[0]

```

```
    base_sample=np.zeros((error_samples,len(bin_sequence)-1))

    for i in range(0,error_samples):
        new_data_sample = np.random.choice(data,samples)
        new_histo_prob, bin_edges = np.histogram(new_data_sample,
        ↪  bin_sequence, density=True)
        base_sample[i,:] = new_histo_prob

    sample_95CI = np.array(2 * (np.std(base_sample,0)))

    return sample_95CI

# This is a helper function for run_simulation that runs bash commands in
↪  separate processes
def execute_command(cmd):
    return subprocess.call(cmd, shell=True)

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

15.1.9 /python_fixtures/Wan_output_fixture.py

```
import multiprocessing
import os
import subprocess
import datetime

import numpy as np
from openpyxl.styles.builtins import output
from sklearn.datasets.tests.test_svmlight_format import currdir

executable = '/home/main/chaste_build/projects/ISP/apps/WanSimulator'

if not(os.path.isfile(executable)):
    raise Exception('Could not find executable: ' + executable)

#########################
# SIMULATION PARAMETERS
#########################

#Define start and end RNG seeds; determines:
#No. simulated CMZs per run
```

```
#unique sequence of RNG results for each lineage
start_seed = 0
end_seed = 99 #total seeds should be divisible by # cores

output_directory = "WanOutput"

##########################
#MODEL PARAMETERS
##########################

#CMZ residency time
wan_residency_time = 17.0
#factor to divide 3dpf progenitor population by to obtain estimate of stem cell population
stem_divisor = 10
#3dpf CMZ progenitor population mean and standard deviation
progenitor_mean = 792
progenitor_std = 160

#stem cell cycle parameters- cell cycle RV is a shifted gamma distribution
stem_gamma_shift = 4
stem_gamma_shape = 6.5 #mean 30 hr stem cell time - in reality is probably more like 60+
stem_gamma_scale = 4

progenitor_gamma_shift = 4  #default He et al. values, mean 6 hr cycle time
progenitor_gamma_shape = 2
progenitor_gamma_scale = 1
progenitor_gamma_sister = 1

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)

#STOCHASTIC MITOTIC MODE
#HE ORIGINAL PARAMETERS
#mitotic mode per-phase probabilities
#3-phase mitotic mode time periodisation
```

```
mitotic_mode_phase_2 = 8 #These are phase boundaries rather than lengths as in eg. He_output_fixture.py
mitotic_mode_phase_3 = 15
phase_1_pPP = 1.0
phase_1_pPD = 0.0
phase_2_pPP = 0.2
phase_2_pPD = 0.4
phase_3_pPP = 0.2
phase_3_pPD = 0.0

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)

def main():

    command_list = []

    # Use processes equal to the number of cpus available
    cpu_count = multiprocessing.cpu_count()
    seeds_per_command = (end_seed + 1) / cpu_count
    base_command = executable + " " + output_directory

    for i in range(0,cpu_count):
        curr_start_seed = i * seeds_per_command
        curr_end_seed = curr_start_seed + (seeds_per_command - 1)

        command = base_command + " "\
            +str(curr_start_seed) + " "\
            +str(curr_end_seed) + " "\
            +cmz_theta_string + " "\
            +stochastic_theta_string

        #command_list.append(command)


        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
95      print("Starting simulations with " + str(cpu_count) + " processes")
96
97      # Generate a pool of workers
98      pool = multiprocessing.Pool(processes=cpu_count)
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__":
```

```
    main()
```

15.1.10 /python_fixtures/diagram_utility_scripts/Gomes_He_cycle_plots.py

```
import os
import numpy as np
import matplotlib.pyplot as plt
from fileinput import filename
from scipy.stats import lognorm
from scipy.stats import gamma
from statsmodels.sandbox.distributions.gof_new import a_st70_upp

#Model parameters
gomes_normal_sigma = .32839
gomes_normal_mu = 3.9716

he_gamma_shape = 2
he_gamma_scale = 1
he_gamma_offset = 4

def main():
    gomes_x_range = np.arange(1,151,1)
    gomes_y = lognorm.pdf(gomes_x_range, gomes_normal_sigma, 0,
    ↪  np.exp(gomes_normal_mu))

    he_x_range = np.arange(1,16,.1)
    he_y = gamma.pdf(he_x_range, he_gamma_shape, he_gamma_offset,
    ↪  he_gamma_scale)

    fig, (ax_g, ax_h) = plt.subplots(1,2,figsize=(6,2))

    ax_g.plot(gomes_x_range,gomes_y,'k-')
    ax_h.plot(he_x_range, he_y, 'k-')

    plt.sca(ax_g)
    plt.xlabel("Cell cycle duration (h)")
    plt.ylabel("Probability")

    plt.sca(ax_h)
    plt.xlabel("Cell cycle duration (h)")
    plt.ylabel("Probability")
```

```python

    plt.tight_layout()
    plt.savefig('/home/main/Desktop/utility.png', transparent=True)
    plt.show()

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

15.1.11 /python_fixtures/diagram_utility_scripts/He_Boije_signal_plots.py

```python
import os
import numpy as np
import matplotlib.pyplot as plt
from fileinput import filename
from scipy.stats import lognorm
from scipy.stats import gamma
from statsmodels.sandbox.distributions.gof_new import a_st70_upp

#Model parameters
he_x = [0,8,15,25]
he_pp_y = [1,.2,.2,.2]
he_pd_y = [0,.4,0,0]
he_dd_y = [0,.4,.8,.8]

boije_x = [0,3,5,9]
boije_pAtoh7 = [0,.32,0,0]
boije_pPtf1a = [0,.3,0,0]
boije_pNg = [0,0,.8,.8]

def main():
    fig, ((pp,pd,dd))= plt.subplots(1,3,figsize=(6,2),sharey=True)

    fig2, ((pAtoh7),(pPtf1a),(pNg)) = plt.subplots(3,1,figsize=(2,4),sharex=True)

    pp.set_xlim((0,24))
    pd.set_xlim((0,24))
    dd.set_xlim((0,24))

    pp.set_xticks([8,15])
    pd.set_xticks([8,15])
```

```
    dd.set_xticks([8,15])

    pp.set_ylim((-.1,1.1))
    pp.set_yticks(np.arange(0,1.2,.2))
    pp.set_ylabel("Probability")
    pp.set_xlabel("TiL (h)")
    pp.axvline(8,linestyle='--',color='.1', alpha=.2)
    pp.axvline(15,linestyle='--',color='.1', alpha=.2)
    pp.step(he_x,he_pp_y, 'k-', where='post', linewidth=2)

    pd.set_ylim((-.1,1.1))
    pd.set_yticks(np.arange(0,1.2,.2))
    pd.set_xlabel("TiL (h)")
    pd.axvline(8,linestyle='--',color='.1', alpha=.2)
    pd.axvline(15,linestyle='--',color='.1', alpha=.2)
    pd.step(he_x,he_pd_y, 'k-', where='post', linewidth=2)

    dd.set_ylim((-.1,1.1))
    dd.set_yticks(np.arange(0,1.2,.2))
    dd.set_xlabel("TiL (h)")
    dd.axvline(8,linestyle='--',color='.1', alpha=.2)
    dd.axvline(15,linestyle='--',color='.1', alpha=.2)
    dd.step(he_x,he_dd_y, 'k-', where='post', linewidth=2)

    pAtoh7.set_xlim((0,8))
    pNg.set_xlim((0,8))
    pPtf1a.set_xlim((0,8))

    pAtoh7.set_xticks([3,5])
    pNg.set_xticks([3,5])
    pPtf1a.set_xticks([3,5])


    pNg.set_xlabel("Generation")


    pAtoh7.set_ylim((-.1,1.1))
    pAtoh7.set_yticks(np.arange(0,1.25,.25))
    pAtoh7.set_ylabel("Probability")
    pAtoh7.axvline(3,linestyle='--',color='.1',alpha=.2)
    pAtoh7.axvline(5,linestyle='--',color='.1',alpha=.2)
    pAtoh7.step(boije_x,boije_pAtoh7, 'k-', where='post', linewidth=2)

```

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

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


    plt.tight_layout()
    fig.savefig('Hemode.png', transparent=True)
    fig2.savefig('Boijesignals.png', transparent=True)
    plt.show()

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

15.1.12 /python_fixtures/figure_plots/Cumulative_EdU.py

```
import os
import numpy as np
import statsmodels.api as sm
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from PIL import Image
from io import BytesIO


#PLoS formatting stuff
plt.rcParams['font.size'] = 12
plt.rcParams['font.family'] = 'sans-serif'
plt.rcParams['font.sans-serif'] = ['Arial']

def main():
```

```
#Read & parse raw data file
raw_data = pd.read_excel('/home/main/git/chaste/projects/ISP/empirical_data/cumulative_e
#group data by dorsal/ventral
dv_grouped = raw_data.groupby('D/V')
#make new dataframe for totals and labelled fraction
totals_frame = dv_grouped.get_group('D')
#reset indices to ventral frame for addition operatinos
totals_frame = totals_frame.set_index(dv_grouped.get_group('V').index)
#sum D + V PCNA & EdU counts
totals_frame.PCNA = totals_frame.PCNA + dv_grouped.get_group('V').PCNA
totals_frame.EdU = totals_frame.EdU + dv_grouped.get_group('V').EdU
#create new column for labelled fraction
totals_frame['labelled_fraction'] = totals_frame.EdU / totals_frame.PCNA

#setup x for linear regression with y-intercept constant
X = totals_frame.Time
X = sm.add_constant(X)

model = sm.OLS(totals_frame.labelled_fraction, X).fit()
print(model.summary())
fit_results = model.params

tc = 1/fit_results[1]
ts = tc * fit_results[0]

sns.regplot(totals_frame.Time,totals_frame.labelled_fraction)

plt.text(7,.2,"Tc: " + f'{tc:.2f}')
plt.text(7,.1,"Ts: " + f'{ts:.2f}')

plt.xlabel("Time (h)")
plt.ylabel("Fraction of CMZ RPCs labelled by EdU")

plt.savefig("cumulative_edu.png")
png_memory = BytesIO()
plt.savefig(png_memory, format='png', dpi=600)
PILpng = Image.open(png_memory)
PILpng.save('cumulative_edu.tiff')
png_memory.close()
```

```

    plt.show()

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

15.1.13 /python_fixtures/figure_plots/He_output_plot.py

```
import os
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from fileinput import filename
from statsmodels.sandbox.distributions.gof_new import a_st70_upp

from PIL import Image
from io import BytesIO

#PLoS formatting stuff
plt.rcParams['font.size'] = 10
plt.rcParams['font.family'] = 'sans-serif'
plt.rcParams['font.sans-serif'] = ['Arial']

#AIC & Plotting utility params
count_seeds = 10000
event_seeds = 1000
error_samples = 5000 #number of samples to draw when estimating
 ↪  plausibility interval for simulations

he_model_params = 15
det_model_params = 13

#stats & line_plotter utility arrays
bin_sequence_24_32 = np.arange(1,26,1)
x_sequence_24_32 = np.arange(1,25,1)

bin_sequence_48 = np.arange(1,11,1)
x_sequence_48 = np.arange(1,10,1)

bin_sequence_wan = np.arange(2,17,1)
x_sequence_wan = np.arange(2,16,1)

```

```
bin_sequence_events = np.arange(30,85,5)
x_sequence_events = np.arange(30,80,5)

##############################
# HE ET AL EMPIRICAL RESULTS
##############################

#Count probability arrays
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
raw_counts_24 = raw_counts[0:64,:]
raw_counts_32 = raw_counts[64:233,:]
raw_counts_48 = raw_counts[233:396,:]

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
prob_empirical_32,bin_edges = np.histogram(np.sum(raw_counts_32,axis=1),bin_sequence_24_32,density=True)
prob_empirical_48,bin_edges = np.histogram(np.sum(raw_counts_48,axis=1),bin_sequence_48,density=True)

#no. of lineages EO per induction timepoint/event group
lineages_sampled_24 = 64
lineages_sampled_32 = 169
lineages_sampled_48 = 163
lineages_sampled_events = 60

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

EO_PP = EO_events[np.where(EO_events[:,0]==0)]
EO_PD = EO_events[np.where(EO_events[:,0]==1)]
EO_DD = EO_events[np.where(EO_events[:,0]==2)]

```

```
prob_empirical_PP,bin_edges = np.histogram(EO_PP,bin_sequence_events,density=True)
prob_empirical_PD,bin_edges = np.histogram(EO_PD,bin_sequence_events,density=True)
prob_empirical_DD,bin_edges = np.histogram(EO_DD,bin_sequence_events,density=True)

empirical_fit_list = [prob_empirical_24,prob_empirical_32,prob_empirical_48,prob_empirical_PP,prob_emp

#mutant results
wt_mean_clone_size = 6
ath5_mean_clone_size = 8
wt_even_odd_ratio = 1.1
ath5_even_odd_ratio = 2.2

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

empirical_test_list = [prob_wan, wt_mean_clone_size, ath5_mean_clone_size, wt_even_odd_ratio, ath5_even_odd_ratio]

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

```
    o_events_PD = np.array(o_events[np.where(o_events[:,1]==1)])
    o_events_DD = np.array(o_events[np.where(o_events[:,1]==2)])
    o_counts_wan_no_shadow = np.loadtxt("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/HeC skiprows=1, usecols=3)
    o_counts_ath5 = np.loadtxt("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/HeC skiprows=1, usecols=3)
    o_counts_validate = np.loadtxt("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/HeC skiprows=1, usecols=3)

    #refit stochastic mitotic mode files
    s_counts_24 = np.loadtxt("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/HeC skiprows=1, usecols=3)
    s_counts_32 = np.loadtxt("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/HeC skiprows=1, usecols=3)
    s_counts_48 = np.loadtxt("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/HeC skiprows=1, usecols=3)
    s_events = np.loadtxt("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/HeM skiprows=1, usecols=(0,3))
    s_events_PP = np.array(s_events[np.where(s_events[:,1]==0)])
    s_events_PD = np.array(s_events[np.where(s_events[:,1]==1)])
    s_events_DD = np.array(s_events[np.where(s_events[:,1]==2)])
    s_counts_wan_no_shadow = np.loadtxt("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/HeC skiprows=1, usecols=3)
    s_counts_ath5 = np.loadtxt("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/HeC skiprows=1, usecols=3)
    s_counts_validate = np.loadtxt("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/HeC skiprows=1, usecols=3)

    #deterministic mitotic mode files
    d_counts_24 = np.loadtxt("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/HeC skiprows=1, usecols=3)
```

```
d_counts_32 = np.loadtxt("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/HeC
skiprows=1, usecols=3)
d_counts_48 = np.loadtxt("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/HeC
skiprows=1, usecols=3)
d_events = np.loadtxt("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/HeM
skiprows=1, usecols=(0,3))
d_events_PP = np.array(d_events[np.where(d_events[:,1]==0)])
d_events_PD = np.array(d_events[np.where(d_events[:,1]==1)])
d_events_DD = np.array(d_events[np.where(d_events[:,1]==2)])
d_counts_wan_no_shadow = np.loadtxt("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/HeC
skiprows=1, usecols=3)
d_counts_ath5 = np.loadtxt("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/HeC
skiprows=1, usecols=3)
d_counts_validate = np.loadtxt("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/HeC
skiprows=1, usecols=3)

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

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

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

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

s_wt_even_odd_ratio = len(np.where(s_counts_validate % 2 == 0)[0])/len(np.where(s_counts_validate % 2 == 1)[0])
s_ath5_even_odd_ratio = len(np.where(s_counts_ath5 % 2 == 0)[0])/len(np.where(s_counts_ath5 % 2 == 1)[0])
```

```
    d_wt_even_odd_ratio = len(np.where(d_counts_validate % 2 == 0)[0])/len(np.where(d_counts_validate % 2 == 1)[0])
    d_ath5_even_odd_ratio = len(np.where(d_counts_ath5 % 2 == 0)[0])/len(np.where(d_counts_ath5 % 2 == 1)[0])

    o_fit_output_list = [o_counts_24, o_counts_32, o_counts_48, o_events_PP, o_events_PD, o_events_DD]
    o_test_output_list = [o_counts_wan_no_shadow, o_wt_mean_clone_size, o_ath5_mean_clone_size, o_wt_even_odd_ratio, o_ath5_even_odd_ratio]

    s_fit_output_list = [s_counts_24, s_counts_32, s_counts_48, s_events_PP, s_events_PD, s_events_DD]
    s_test_output_list = [s_counts_wan_no_shadow, s_wt_mean_clone_size, s_ath5_mean_clone_size, s_wt_even_odd_ratio, s_ath5_even_odd_ratio]

    d_fit_output_list = [d_counts_24, d_counts_32, d_counts_48, d_events_PP, d_events_PD, d_events_DD]
    d_test_output_list = [d_counts_wan_no_shadow, d_wt_mean_clone_size, d_ath5_mean_clone_size, d_wt_even_odd_ratio, d_ath5_even_odd_ratio]

    #SETUP FIGURES
    #4 figures to be generated; 3 show output of individual model modes, 4th overlays stochastic refit on deterministic fit
    o_fig, ((o_24,o_32),(o_48,o_PP),(o_PD,o_DD),(o_wan_noshad, o_ath5size),(o_ratio,o_unused)) = 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, s_ath5size),(s_ratio,s_unused)) = 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, d_ath5size),(d_ratio,d_unused)) = 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, sd_ath5size),(sd_ratio,sd_unused)) = plt.subplots(5,2,figsize=(7.5,8.75))

    #turn off unused axes

    o_unused.axis('off');s_unused.axis('off');d_unused.axis('off');sd_unused.axis

```

```
#set xlabels
o_24.set_xlabel('Count');o_32.set_xlabel('Count');o_48.set_xlabel('Count');o_PP.set_xlabel('Age (hpf)');o_PD.set_xlabel('Age (hpf)');o_DD.set_xlabel('Age (hpf)')

s_24.set_xlabel('Count');s_32.set_xlabel('Count');s_48.set_xlabel('Count');s_PP.set_xlabel('Age (hpf)');s_PD.set_xlabel('Age (hpf)');s_DD.set_xlabel('Age (hpf)')

d_24.set_xlabel('Count');d_32.set_xlabel('Count');d_48.set_xlabel('Count');d_PP.set_xlabel('Age (hpf)');d_PD.set_xlabel('Age (hpf)');d_DD.set_xlabel('Age (hpf)')

sd_24.set_xlabel('Count');sd_32.set_xlabel('Count');sd_48.set_xlabel('Count')sd_PP.set_xlabel('Age (hpf)');sd_PD.set_xlabel('Age (hpf)');sd_DD.set_xlabel('Age (hpf)')

#set ylabels
o_24.set_ylabel('Probability');o_32.set_ylabel('Probability');o_48.set_ylabelo_PP.set_ylabel('PP Probability');o_PD.set_ylabel('PD Probability');o_DD.set_ylabel('DD Probability');o_ath5size.set_ylabel('Clone size');o_ratio.set_ylabel('Clone even/odd ratio')

s_24.set_ylabel('Probability');s_32.set_ylabel('Probability');s_48.set_ylabels_PP.set_ylabel('PP Probability');s_PD.set_ylabel('PD Probability');s_DD.set_ylabel('DD Probability');s_ath5size.set_ylabel('Clone size');s_ratio.set_ylabel('Clone even/odd ratio')

d_24.set_ylabel('Probability');d_32.set_ylabel('Probability');d_48.set_ylabeld_PP.set_ylabel('PP Probability');d_PD.set_ylabel('PD Probability');d_DD.set_ylabel('DD Probability');d_ath5size.set_ylabel('Clone size');d_ratio.set_ylabel('Clone 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
192 #PLOT DATA
193 #plot lines, CIs, and bars
194 o_objects = line_plotter(o_24, o_counts_24, bin_sequence_24_32,
    ↪ x_sequence_24_32, count_seeds, lineages_sampled_24,
    ↪ prob_empirical_24, 'b-', '#000080', '#0000FF', 'y+', 0)
195 line_plotter(o_32, o_counts_32, bin_sequence_24_32, x_sequence_24_32,
    ↪ count_seeds, lineages_sampled_32, prob_empirical_32, 'b-',
    ↪ '#000080', '#0000FF', 'y+', 0)
196 line_plotter(o_48, o_counts_48, bin_sequence_48, x_sequence_48,
    ↪ count_seeds, lineages_sampled_48, prob_empirical_48, 'b-',
    ↪ '#000080', '#0000FF', 'y+', 0)
197 line_plotter(o_PP, o_events_PP[:,0], bin_sequence_events,
    ↪ x_sequence_events, event_seeds,
    ↪ lineages_sampled_events,prob_empirical_PP,'b-', '#000080',
    ↪ '#0000FF', 'y+', 1)
198 line_plotter(o_PD, o_events_PD[:,0], bin_sequence_events,
    ↪ x_sequence_events, event_seeds,
    ↪ lineages_sampled_events,prob_empirical_PD,'b-', '#000080',
    ↪ '#0000FF', 'y+', 1)
199 line_plotter(o_DD, o_events_DD[:,0], bin_sequence_events,
    ↪ x_sequence_events, event_seeds,
    ↪ lineages_sampled_events,prob_empirical_DD,'b-', '#000080',
    ↪ '#0000FF', 'y+', 1)
200 line_plotter(o_wan_noshad, o_counts_wan_no_shadow, bin_sequence_wan,
    ↪ x_sequence_wan, count_seeds, lineages_sampled_wan, prob_wan,
    ↪ 'b-', '#000080', '#0000FF', 'y+', 2)
201 o_ath5size.bar([0,1,2,3], [(wt_mean_clone_size/wt_mean_clone_size),
    ↪ (o_wt_mean_clone_size/wt_mean_clone_size),
    ↪ (ath5_mean_clone_size/wt_mean_clone_size),
    ↪ (o_ath5_mean_clone_size/wt_mean_clone_size)],color=['#000000','#000080','#000
    ↪ WT','OSM WT', 'EO Ath5mo', 'OSM Ath5mo'])
202 o_ratio.bar([0,1,2,3], [wt_even_odd_ratio, o_wt_even_odd_ratio,
    ↪ ath5_even_odd_ratio,
    ↪ o_ath5_even_odd_ratio],color=['#000000','#000080','#000000','#000080'],edgeco
    ↪ WT','OFit WT', 'EO Ath5mo', 'OFit Ath5mo'])
203 plt.setp(o_ath5size.get_xticklabels(), rotation=25, ha='right')
```

```
plt.setp(o_ratio.get_xticklabels(), rotation=25, ha='right')

s_objects = line_plotter(s_24, s_counts_24, bin_sequence_24_32, x_sequence_24_32, count_seeds, lineages_sampled_24, prob_empirical_24, 'm-', '#800080', '#FF00FF', 'k+', 0)
line_plotter(s_32, s_counts_32, bin_sequence_24_32, x_sequence_24_32, count_seeds, lineages_sampled_32, prob_empirical_32, 'm-', '#800080', '#FF00FF', 'k+', 0)
line_plotter(s_48, s_counts_48, bin_sequence_48, x_sequence_48, count_seeds, lineages_sampled_48, prob_empirical_48, 'm-', '#800080', '#FF00FF', 'k+', 0)
line_plotter(s_PP, s_events_PP[:,0], bin_sequence_events, x_sequence_events, event_seeds, lineages_sampled_events,prob_empirical_PP,'m-', '#800080', '#FF00FF', 'k+', 1)
line_plotter(s_PD, s_events_PD[:,0], bin_sequence_events, x_sequence_events, event_seeds, lineages_sampled_events,prob_empirical_PD,'m-', '#800080', '#FF00FF', 'k+', 1)
line_plotter(s_DD, s_events_DD[:,0], bin_sequence_events, x_sequence_events, event_seeds, lineages_sampled_events,prob_empirical_DD,'m-', '#800080', '#FF00FF', 'k+', 1)
line_plotter(s_wan_noshad, s_counts_wan_no_shadow, bin_sequence_wan, x_sequence_wan, count_seeds, lineages_sampled_wan, prob_wan, 'm-', '#800080', '#FF00FF', 'k+', 2)
s_ath5size.bar([0,1,2,3], [(wt_mean_clone_size/wt_mean_clone_size), (s_wt_mean_clone_size/wt_mean_clone_size),(ath5_mean_clone_size/wt_mean_clone
'#000000', '#800080'],edgecolor='#000000',tick_label=['EO WT','SM WT', 'EO Ath5mo', 'SM Ath5mo'])
s_ratio.bar([0,1,2,3], [wt_even_odd_ratio, s_wt_even_odd_ratio, ath5_even_odd_ratio, s_ath5_even_odd_ratio],color=['#000000','#000080','#000000','#000080'],edgeco
WT','SM WT', 'EO Ath5mo', 'SM Ath5mo'])
plt.setp(s_ath5size.get_xticklabels(), rotation=25, ha='right')
plt.setp(s_ratio.get_xticklabels(), rotation=25, ha='right')


d_objects = line_plotter(d_24, d_counts_24, bin_sequence_24_32, x_sequence_24_32, count_seeds, lineages_sampled_24, prob_empirical_24, 'g-', '#008000', '#00FF00', 'k+', 0)
```

```
line_plotter(d_32, d_counts_32, bin_sequence_24_32, x_sequence_24_32, count_seeds, lineages_sampled_32, prob_empirical_32, 'g-', '#008000', '#00FF00', 'k+', 0)
line_plotter(d_48, d_counts_48, bin_sequence_48, x_sequence_48, count_seeds, lineages_sampled_48, prob_empirical_48, 'g-', '#008000', '#00FF00', 'k+', 0)
line_plotter(d_PP, d_events_PP[:,0], bin_sequence_events, x_sequence_events, event_seeds, lineages_sampled_events,prob_empirical_PP,'g-', '#008000', '#00FF00', 'k+', 1)
line_plotter(d_PD, d_events_PD[:,0], bin_sequence_events, x_sequence_events, event_seeds, lineages_sampled_events,prob_empirical_PD,'g-', '#008000', '#00FF00', 'k+', 1)
line_plotter(d_DD, d_events_DD[:,0], bin_sequence_events, x_sequence_events, event_seeds, lineages_sampled_events,prob_empirical_DD,'g-', '#008000', '#00FF00', 'k+', 1)
line_plotter(d_wan_noshad, d_counts_wan_no_shadow, bin_sequence_wan, x_sequence_wan, count_seeds, lineages_sampled_wan, prob_wan, 'g-', '#008000', '#00FF00', 'k+', 2)
d_ath5size.bar([0,1,2,3], [(wt_mean_clone_size/wt_mean_clone_size), (d_wt_mean_clone_size/wt_mean_clone_size),(ath5_mean_clone_size/wt_mean_clone
WT','DM WT', 'EO Ath5mo', 'DM Ath5mo'])
d_ratio.bar([0,1,2,3], [wt_even_odd_ratio, d_wt_even_odd_ratio, ath5_even_odd_ratio, d_ath5_even_odd_ratio],color=['#000000','#008000','#000000','#008000'],edgeco
WT','DM WT', 'EO Ath5mo', 'DM Ath5mo'])
plt.setp(d_ath5size.get_xticklabels(), rotation=25, ha='right')
plt.setp(d_ratio.get_xticklabels(), rotation=25, ha='right')


sd_objects_s = line_plotter(sd_24, s_counts_24, bin_sequence_24_32, x_sequence_24_32, count_seeds, lineages_sampled_24, prob_empirical_24, 'm-', '#800080', '#FF00FF', 'k+', 0)
sd_objects_d = line_plotter(sd_24, d_counts_24, bin_sequence_24_32, x_sequence_24_32, count_seeds, lineages_sampled_24, prob_empirical_24, 'g-', '#008000', '#00FF00', 'k+', 0)
line_plotter(sd_32, s_counts_32, bin_sequence_24_32, x_sequence_24_32, count_seeds, lineages_sampled_32, prob_empirical_32, 'm-', '#800080', '#FF00FF', 'k+', 0)
```

```
    line_plotter(sd_32, d_counts_32, bin_sequence_24_32, x_sequence_24_32, count_seeds, lineages_sampled_32, prob_empirical_32, 'g-', '#008000', '#00FF00', 'k+', 0)
    line_plotter(sd_48, s_counts_48, bin_sequence_48, x_sequence_48, count_seeds, lineages_sampled_48, prob_empirical_48, 'm-', '#800080', '#FF00FF', 'k+', 0)
    line_plotter(sd_48, d_counts_48, bin_sequence_48, x_sequence_48, count_seeds, lineages_sampled_48, prob_empirical_48, 'g-', '#008000', '#00FF00', 'k+', 0)
    line_plotter(sd_PP, s_events_PP[:,0], bin_sequence_events, x_sequence_events, event_seeds, lineages_sampled_events,prob_empirical_PP,'m-', '#800080', '#FF00FF', 'k+', 1)
    line_plotter(sd_PP, d_events_PP[:,0], bin_sequence_events, x_sequence_events, event_seeds, lineages_sampled_events,prob_empirical_PP,'g-', '#008000', '#00FF00', 'k+', 1)
    line_plotter(sd_PD, s_events_PD[:,0], bin_sequence_events, x_sequence_events, event_seeds, lineages_sampled_events,prob_empirical_PD,'m-', '#800080', '#FF00FF', 'k+', 1)
    line_plotter(sd_PD, d_events_PD[:,0], bin_sequence_events, x_sequence_events, event_seeds, lineages_sampled_events,prob_empirical_PD,'g-', '#008000', '#00FF00', 'k+', 1)
    line_plotter(sd_DD, s_events_DD[:,0], bin_sequence_events, x_sequence_events, event_seeds, lineages_sampled_events,prob_empirical_DD,'m-', '#800080', '#FF00FF', 'k+', 1)
    line_plotter(sd_DD, d_events_DD[:,0], bin_sequence_events, x_sequence_events, event_seeds, lineages_sampled_events,prob_empirical_DD,'g-', '#008000', '#00FF00', 'k+', 1)
    line_plotter(sd_wan_noshad, s_counts_wan_no_shadow, bin_sequence_wan, x_sequence_wan, count_seeds, lineages_sampled_wan, prob_wan, 'm-', '#800080', '#FF00FF', 'k+', 2)
    line_plotter(sd_wan_noshad, d_counts_wan_no_shadow, bin_sequence_wan, x_sequence_wan, count_seeds, lineages_sampled_wan, prob_wan, 'g-', '#008000', '#00FF00', 'k+', 2)
    sd_ath5size.bar([0,1,2,3,4,5], [(wt_mean_clone_size), (s_wt_mean_clone_size),(d_wt_mean_clone_size),(ath5_mean_clone_size),(s_ath5_
    WT','SM WT','DM WT', 'EO Ath5mo', 'SM Ath5Mo', 'DM Ath5mo'])
```

```
    sd_ratio.bar([0,1,2,3,4,5], [wt_even_odd_ratio, s_wt_even_odd_ratio, d_wt_even_odd_ratio, ath5_even_odd_ratio, s_ath5_even_odd_ratio, d_ath5_even_odd_ratio],color=['#000000','#800080','#008000','#000000','#80008
    WT','SM WT','DM WT', 'EO Ath5mo', 'SM Ath5Mo', 'DM Ath5mo'])
    plt.setp(sd_ath5size.get_xticklabels(), rotation=25, ha='right')
    plt.setp(sd_ratio.get_xticklabels(), rotation=25, ha='right')

    #calculate AICs, write table to file
    o_fit_AIC = calculate_fit_AIC(o_fit_output_list, empirical_fit_list, he_model_params)
    o_test_AIC = calculate_test_AIC(o_test_output_list, empirical_fit_list, he_model_params)

    s_fit_AIC = calculate_fit_AIC(s_fit_output_list, empirical_fit_list, he_model_params)
    s_test_AIC = calculate_test_AIC(s_test_output_list, empirical_test_list, he_model_params)

    d_fit_AIC = calculate_fit_AIC(d_fit_output_list, empirical_fit_list, det_model_params)
    d_test_AIC = calculate_test_AIC(d_test_output_list, empirical_test_list, det_model_params)

    AIC_table('AIC_table.tex', o_fit_AIC, o_test_AIC, s_fit_AIC, s_test_AIC, d_fit_AIC, d_test_AIC)

    #plot annotiations
    plot_annotater(s_fig,s_objects,'Stochastic Model')
    plot_annotater(o_fig,o_objects,'Stochastic Model (original fit)')
    plot_annotater(d_fig,d_objects, 'Deterministic Model')
    plot_annotater(sd_fig,sd_objects_s, 'Stochastic Model', sd_objects_d, 'Deterministic Model')

    #export as .tiff, .png
    o_fig.savefig('o_fig.png',dpi=600)
    png_memory = BytesIO()
    o_fig.savefig(png_memory, format='png', dpi=600)
    PILpng = Image.open(png_memory)
    PILpng.save('o_fig.tiff')
    png_memory.close()

    s_fig.savefig('s_fig.png',dpi=600)
    png_memory = BytesIO()
```

```
    s_fig.savefig(png_memory, format='png', dpi=600)
    PILpng = Image.open(png_memory)
    PILpng.save('s_fig.tiff')
    png_memory.close()

    d_fig.savefig('d_fig.png',dpi=600)
    png_memory = BytesIO()
    d_fig.savefig(png_memory, format='png', dpi=600)
    PILpng = Image.open(png_memory)
    PILpng.save('d_fig.tiff')
    png_memory.close()

    sd_fig.savefig('sd_fig.png',dpi=600)
    png_memory = BytesIO()
    sd_fig.savefig(png_memory, format='png', dpi=600)
    PILpng = Image.open(png_memory)
    PILpng.save('sd_fig.tiff')
    png_memory.close()

    plt.show()


def calculate_fit_AIC(output_list, empirical_list, number_params):
    #function calculates AIC for He et al counts & event probability density (NOT per-lineage event probability)
    #this numpy array holds the individual RSS vals for timepoints + events
    rss = np.zeros(6)
    number_comparisons = 0

    for i in range(0,3):
        output_counts = output_list[i]
        empirical_prob = empirical_list[i]

        output_histo,bin_edges = np.histogram(output_counts, np.arange(1,len(empirical_prob)+2), density = False)
        output_prob = np.array(output_histo/count_seeds)

        residual = np.array(output_prob- empirical_prob)
        number_comparisons += len(residual)
        rss[i] = np.sum(np.square(residual))

    for i in range (0,3):
```

```
        output_events = output_list[i+3]
        empirical_prob = empirical_list[i+3]

        histo_events,bin_edges = np.histogram(output_events,
         ↪  bin_sequence_events, density=True)
        output_prob = np.array(histo_events/event_seeds)

        residual = np.array(output_prob - empirical_prob)
        number_comparisons += len(residual)
        rss[i+3] = np.sum(np.square(residual))

    AIC = 2 * number_params + number_comparisons * np.log(np.sum(rss))

    return AIC

def calculate_test_AIC(output_list, empirical_list, number_params):
    #function calculates AIC for He mutant data & Wan counts
    rss=np.zeros(3)
    number_comparisons = 0

    #Wan data
    for i in range (0,1):
        output_counts = output_list[i]
        empirical_prob = empirical_list[i]

        output_histo,bin_edges = np.histogram(output_counts,
         ↪  np.arange(2,len(empirical_prob)+3), density = False)
        output_prob = np.array(output_histo/count_seeds)

        residual = np.array(output_prob- empirical_prob)
        number_comparisons += len(residual)
        rss[i] = np.sum(np.square(residual))

    #He data
    for i in range(1,3):
        output_measure = output_list[i]
        empirical_measure = empirical_list[i]

        residual = np.array(output_measure - empirical_measure)
        number_comparisons += 1
        rss[i] = np.sum(np.square(residual))

    AIC = 2 * number_params + number_comparisons * np.log(np.sum(rss))
```

```python

    return AIC

#line plotter plots average counts or event event +- 2 standard deviations (He et al's "95% probability interval")
#"mode" 1 gives correct hourly output for 5-hour event bins. mode 2 begins Wan plots at clone size 2, reflecting unavailable data
def line_plotter(subplot, data, bin_sequence, x_sequence, seeds, samples, empirical_prob, line_colour_string, fill_edge_colour_string, fill_face_colour_string, empirical_colour_string, mode):

    legend_objects = []

    prob_histo, bin_edges = np.histogram(data, bin_sequence, density=True)

    #estimate of 95% CI for empirical observations of model output from repeated sampling of the data w/ the appropriate number of empirically observed lineages
    interval = sampler(data,samples,bin_sequence)
    legend_objects.append(subplot.plot(x_sequence,prob_histo, line_colour_string))
    subplot.fill_between(x_sequence, (prob_histo - interval), (prob_histo + interval), alpha=0.1, edgecolor=fill_edge_colour_string, facecolor=fill_face_colour_string)

    if mode == 0:

        legend_objects.append(subplot.plot(np.arange(1,len(empirical_prob)+1,1),e
        empirical_colour_string))

    if mode == 1:

        legend_objects.append(subplot.plot(np.arange(30,80,5),empirical_prob, empirical_colour_string))

    if mode == 2:

        legend_objects.append(subplot.plot(np.arange(2,len(empirical_prob)+2,1), empirical_prob, empirical_colour_string))

    return legend_objects

```

```
def sampler(data,samples,bin_sequence):

    if len(data) == 0: #catches edge case w/ no entries for a mitotic mode
        data=[0]

    base_sample=np.zeros((error_samples,len(bin_sequence)-1))

    for i in range(0,error_samples):
        new_data_sample = np.random.choice(data,samples)
        new_histo_prob, bin_edges = np.histogram(new_data_sample, bin_sequence, density=True)
        base_sample[i,:] = new_histo_prob

    sample_95CI = np.array(2 * (np.std(base_sample,0)))

    return sample_95CI

def plot_annotater(fig_to_ann, legend_objects_1, legend_objects_1_name, legend_objects_2=None, legend_objects_2_name=None):
    #subplot labels
    labels = ['A','B','C','D','E','F','G','H','I','']

    for i, ax in enumerate(fig_to_ann.axes):
        ax.text(.01,.95, labels[i], transform=ax.transAxes, fontweight='bold', va='top')

        time_rectangle_x = .70
        time_rectangle_y = .65
        time_rectangle_width = .2
        time_rectangle_height = .03
        time_caret_half_width = .02
        time_caret_height = np.sqrt(3)*(time_caret_half_width*2)

        if i < 3:
            #add the time bar and label 72 hr caret on appropriate plots

            ax.add_patch(plt.Rectangle((time_rectangle_x,time_rectangle_y),time_r
            color='k', transform=ax.transAxes))

            ax.add_patch(plt.Polygon(((time_rectangle_x+time_rectangle_width,time
            fill=False, transform=ax.transAxes))
```

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

        if i == 0:
            #add 24hr caret

                ax.add_patch(plt.Polygon(((time_rectangle_x,time_rectangle_y+time_rec
                fill=False, transform=ax.transAxes))

                ax.text(time_rectangle_x,time_rectangle_y+time_rectangle_height+time_
                transform=ax.transAxes,
                fontsize='8',va='bottom',ha='center')

        if i == 1:
            #add 32hr caret

                ax.add_patch(plt.Polygon(((time_rectangle_x+(8/48)*time_rectangle_wid
                fill=False, transform=ax.transAxes))

                ax.text(time_rectangle_x+(8/48)*time_rectangle_width,time_rectangle_y
                transform=ax.transAxes,
                fontsize='8',va='bottom',ha='center')

        if i == 2:

                ax.add_patch(plt.Polygon(((time_rectangle_x+(24/48)*time_rectangle_wi
                fill=False, transform=ax.transAxes))

                ax.text(time_rectangle_x+(24/48)*time_rectangle_width,time_rectangle_
                transform=ax.transAxes,
                fontsize='8',va='bottom',ha='center')

        parent_circle_coord = (.85,.8)
        child_1_coord = (.8,.6)
        child_2_coord = (.9,.6)
        ellipse_width = .03
        ellipse_height = 2.5*ellipse_width

        if i == 3:

                ax.add_patch(plt.Polygon(((parent_circle_coord),child_1_coord),color=
```

```
                ↪ ax.add_patch(plt.Polygon(((parent_circle_coord),child_2_coord),color=
                ↪ ax.add_patch(patches.Ellipse(parent_circle_coord,ellipse_width,ellips
                ↪ color='k', transform=ax.transAxes))
                ↪ ax.add_patch(patches.Ellipse(child_1_coord,ellipse_width,ellipse_heig
                ↪ color='k', transform=ax.transAxes))
                ↪ ax.add_patch(patches.Ellipse(child_2_coord,ellipse_width,ellipse_heig
                ↪ color='k', transform=ax.transAxes))
                ax.text(.91, .7, 'PP', transform=ax.transAxes, va='center',
                ↪ ha='left')

        if i == 4:

                ↪ ax.add_patch(plt.Polygon(((parent_circle_coord),child_1_coord),color=
                ↪ ax.add_patch(plt.Polygon(((parent_circle_coord),child_2_coord),color=
                ↪ ax.add_patch(patches.Ellipse(parent_circle_coord,ellipse_width,ellips
                ↪ color='k', transform=ax.transAxes))
                ↪ ax.add_patch(patches.Ellipse(child_1_coord,ellipse_width,ellipse_heig
                ↪ color='k', transform=ax.transAxes))
                ↪ ax.add_patch(patches.Ellipse(child_2_coord,ellipse_width,ellipse_heig
                ↪ edgecolor='k', facecolor='w', transform=ax.transAxes))
                ax.text(.91, .7, 'PD', transform=ax.transAxes, va='center',
                ↪ ha='left')


        if i == 5:

                ↪ ax.add_patch(plt.Polygon(((parent_circle_coord),child_1_coord),color=
                ↪ ax.add_patch(plt.Polygon(((parent_circle_coord),child_2_coord),color=
                ↪ ax.add_patch(patches.Ellipse(parent_circle_coord,ellipse_width,ellips
                ↪ color='k', transform=ax.transAxes))
                ↪ ax.add_patch(patches.Ellipse(child_1_coord,ellipse_width,ellipse_heig
                ↪ edgecolor='k', facecolor='w', transform=ax.transAxes))
```

```
                ax.add_patch(patches.Ellipse(child_2_coord,ellipse_width,ellipse_heig
                edgecolor='k', facecolor='w', transform=ax.transAxes))
                ax.text(.91, .7, 'DD', transform=ax.transAxes, va='center',
                ha='left')


            if i == 6:
                ax.text(.80,.85,'CMZ', transform=ax.transAxes,
                fontweight='bold', va='top')

    if legend_objects_2 == None:

        fig_to_ann.legend((legend_objects_1[1][0],legend_objects_1[0][0]),('Obser
        bbox_to_anchor=(.59,.10,.1,.1), loc='upper left')
    else:

        fig_to_ann.legend((legend_objects_1[1][0],legend_objects_1[0][0],legend_o
        legend_objects_2_name), bbox_to_anchor=(.59,.1,.1,.1),
        loc='upper left')

    #data group box annotations
    fig_to_ann.patches.extend([plt.Rectangle((0.005,0.41),0.99,0.58,
                                fill=False, color='k', linestyle='-',
                                transform=fig_to_ann.transFigure,
                                figure=fig_to_ann)])
    fig_to_ann.patches.extend([plt.Polygon(((0.005, .4),(.995,.4),
    (.995,.21), (.5,.21), (.5, .01), (.005, .01)),
                                fill=False, color='k', linestyle=':',
                                transform=fig_to_ann.transFigure,
                                figure=fig_to_ann)])

    fig_to_ann.text(.605, .075, 'Training Data', fontsize=12, transform =
    fig_to_ann.transFigure, figure=fig_to_ann)
    fig_to_ann.patches.extend([plt.Rectangle((0.6,0.07),0.143,0.029,
                                fill=False, color='k', linestyle='-',
                                transform=fig_to_ann.transFigure,
                                figure=fig_to_ann)])


    fig_to_ann.text(.605, .025, 'Test Data', fontsize=12, transform =
    fig_to_ann.transFigure, figure=fig_to_ann)
    fig_to_ann.patches.extend([plt.Rectangle((0.6,0.02),0.111,0.029,
```

```
                                    fill=False, color='k', linestyle=':',
                                    transform=fig_to_ann.transFigure,
                                    ↪ figure=fig_to_ann)])

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

def AIC_table (filename, o_training_AIC, o_test_AIC, s_training_AIC,
↪ s_test_AIC, d_training_AIC, d_test_AIC):
    #writes LaTeX tabular snippet with AIC values
    table_file = open(filename, 'w')
    table_file.write(r'\begin{tabular}{|l|l|l|}'+'\n')
    table_file.write(r'\hline'+'\n')
    table_file.write(r'{\bf Model} & {\bf Training AIC} & {\bf Test AIC}
    ↪ \\ \thickhline'+'\n')
    table_file.write(r'Stochastic (He fit) &
    ↪ $'+f'{o_training_AIC:.2f}'+r'$ & $' + f'{o_test_AIC:.2f}' + r'$\\
    ↪ \hline'+'\n')
    table_file.write(r'Stochastic (SPSA fit) &
    ↪ $'+f'{s_training_AIC:.2f}'+r'$ & $' + f'{s_test_AIC:.2f}' + r'$\\
    ↪ \hline'+'\n')
    table_file.write(r'Deterministic (SPSA fit) &
    ↪ $'+f'{d_training_AIC:.2f}'+r'$ & $' + f'{d_test_AIC:.2f}' + r'$\\
    ↪ \hline'+'\n')
    table_file.write(r'\end{tabular}'+'\n')

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

15.1.14 /python_fixtures/figure_plots/Kolmogorov_plot.py

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

#AIC & Plotting utility params
event_seeds = 1000
error_samples = 5000 #number of samples to draw when estimating
↪ plausibility interval for simulations

def main():
    #LOAD & PARSE KOLMOGOROV COMPLEXITY FILES
```

```
    gomes_kol = np.loadtxt('/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/Kol
    he_kol = np.loadtxt('/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/Kol
    he_refit_kol = np.loadtxt('/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/Kol
    deterministic_kol = np.loadtxt('/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/Kol
    boije_kol = np.loadtxt('/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/Kol
    eo_kol = np.loadtxt('/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/Kol

    data_sequence = [gomes_kol,he_kol,he_refit_kol,deterministic_kol,boije_kol,eo_kol]

    plt.violinplot(data_sequence, showextrema=True)

    plt.ylabel ('Estimated Kolmogorov Complexity')

    plt.show()

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

15.1.15 /python_fixtures/figure_plots/Mitotic_rate_plot.py

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

from PIL import Image
from io import BytesIO

#PLoS formatting stuff
plt.rcParams['font.size'] = 12
```

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

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

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

##############################
# HE ET AL EMPIRICAL RESULTS
##############################

rate_bin_sequence = np.arange(30,85,5)
rate_x_sequence = np.arange(30,80,5)
rate_trim_value = 10

#no. of lineages EO per induction timepoint/event group
lineages_sampled_events = 60

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

event_counts, bin_edges = np.histogram(EO_events,bin_sequence_events,density=False)
#hourly values
empirical_events_per_lineage = np.array(event_counts/(lineages_sampled_events*5))

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

    #refit stochastic mitotic mode files
```

```
    s_events = np.loadtxt("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/HeM
    skiprows=1, usecols=(0))

    #deterministic mitotic mode files
    d_events = np.loadtxt("/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/HeM
    skiprows=1, usecols=(0))

    original = line_plotter(o_events,bin_sequence_events, x_sequence_events, event_seeds, lineages_sampled_events, 'b--', '#000080', '#0000FF')
    refit = line_plotter(s_events,bin_sequence_events, x_sequence_events, event_seeds, lineages_sampled_events, 'm-', '#800080', '#FF00FF')
    deterministic = line_plotter(d_events,bin_sequence_events, x_sequence_events, event_seeds, lineages_sampled_events, 'g-', '#008000', '#00FF00')

    empirical = plt.plot(np.arange(30,80,5),empirical_events_per_lineage, 'k+')

    plt.legend((original[0], refit[0], deterministic[0], empirical[0]), ('SM (He fit)', 'SM (SPSA fit)', 'DM', 'Observed'))

    plt.xlabel ('Age (hpf)')
    plt.ylabel ('Probability of mitotic event per lineage per hour')


    plt.savefig("per_lineage_mitotic_rate.png")
    png_memory = BytesIO()
    plt.savefig(png_memory, format='png', dpi=600)
    PILpng = Image.open(png_memory)
    PILpng.save('per_lineage_mitotic_rate.tiff')
    png_memory.close()

    plt.show()

#line plotter plots average counts or event event +- 2 standard deviations (He et al's "95% probability interval")
#"mode" 1 gives correct hourly output for 5-hour event bins. mode 2 begins Wan plots at clone size 2, reflecting unavailable data
```

```
def line_plotter(data, bin_sequence, x_sequence, seeds, samples, line_colour_string, fill_edge_colour_string, fill_face_colour_string):

    histo, bin_edges = np.histogram(data, bin_sequence, density=False)
    prob_histo = np.array(histo / (seeds * 5))

    #estimate of 95% CI for empirical observations of model output from repeated sampling of the data w/ the appropriate number of empirically observed lineages
    interval = sampler(data,samples,bin_sequence)
    line = plt.plot(x_sequence,prob_histo, line_colour_string)
    plt.fill_between(x_sequence, (prob_histo - interval), (prob_histo + interval), alpha=0.1, edgecolor=fill_edge_colour_string, facecolor=fill_face_colour_string)

    return line

def sampler(data,samples,bin_sequence):

    if len(data) == 0: #catches edge case w/ no entries for a mitotic mode
        data=[0]

    base_sample=np.zeros((error_samples,len(bin_sequence)-1))

    for i in range(0,error_samples):
        new_data_sample = np.random.choice(data,samples)
        new_histo, bin_edges = np.histogram(new_data_sample, bin_sequence, density=False)
        new_histo_prob = np.array(new_histo/samples)
        base_sample[i,:] = new_histo_prob

    sample_95CI = np.array(2 * (np.std(base_sample,0)))

    return sample_95CI

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

15.1.16 /python_fixtures/figure_plots/Wan_output_plot.py

```python
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker

from PIL import Image
from io import BytesIO

#PLoS formatting stuff
plt.rcParams['font.size'] = 12
plt.rcParams['font.family'] = 'sans-serif'
plt.rcParams['font.sans-serif'] = ['Arial']

def main():
    wan_output_dir = '/home/main/git/chaste/projects/ISP/python_fixtures/testoutput/WanOutput'
    wan_results_list = []
    for root,dirs,files in os.walk(wan_output_dir):
        for name in files:
            if name == 'celltypes.dat':
                results = np.loadtxt(os.path.join(root,name), usecols=2)
                if len(results) == 8569:
                    wan_results_list.append(results)

    wan_sim_results = np.array(wan_results_list)
    wan_sim_mean = np.mean(wan_sim_results,axis=0)
    wan_sim_95CI = 2*np.std(wan_sim_results,axis=0)
    wan_sim_x_seq = np.arange(72,8641,1)/24

    empirical_pop_mean = np.array([792.0, 768.4, 906.1, 1159.7, 1630.0, 3157.9, 3480.2, 4105.1, 1003.0, 477.2, 438.8088611111])
    empirical_pop_95CI = 2*np.array([160.1, 200.1, 244.5, 477.6, 444.3, 1414.3, 472.1, 1169.7, 422.8, 367.5, 294.8])
    empirical_x_seq = np.array([72,120,192,288,408,552,720,1440,2160,4320,8640])/24

    fig, ax = plt.subplots(1,1)

    empirical = plt.plot(empirical_x_seq, empirical_pop_mean, 'g-')
```

```
    plt.fill_between(empirical_x_seq, (empirical_pop_mean - empirical_pop_95CI), (empirical_pop_mean + empirical_pop_95CI), alpha=0.1, edgecolor='#008000', facecolor='#00FF00')

    wan_simulator = plt.plot(wan_sim_x_seq, wan_sim_mean, 'm-')
    plt.fill_between(wan_sim_x_seq, (wan_sim_mean - wan_sim_95CI), (wan_sim_mean + wan_sim_95CI), alpha=0.1, edgecolor='#800080', facecolor='#FF00FF')

    plt.ylim(bottom=0)
    plt.xlim(left=0)

    plt.ylabel("CMZ population size")
    plt.xlabel("Retina age (dpf)")
    ax.xaxis.set_major_locator(ticker.MultipleLocator(40))
    ax.legend((empirical[0], wan_simulator[0]), ('Observed CMZ population', '"Wan-type" simulated CMZ population'))

    png_memory = BytesIO()
    fig.savefig(png_memory, format='png', dpi=600)
    PILpng = Image.open(png_memory)
    PILpng.save('wan_fig.tiff')
    png_memory.close()

    plt.show()

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

15.1.17 /src/BoijeCellCycleModel.cpp

```
#include "BoijeCellCycleModel.hpp"

BoijeCellCycleModel::BoijeCellCycleModel() :
        AbstractSimpleCellCycleModel(), mOutput(false), mEventStartTime(), mSequenceSampler(false), mSeqSamplerLabelSister(
                false), mDebug(false), mTimeID(), mVarIDs(), mDebugWriter(), mGeneration(0), mPhase2gen(3), mPhase3gen(
                5), mprobAtoh7(0.32), mprobPtf1a(0.30), mprobng(0.80), mAtoh7Signal(false), mPtf1aSignal(false), mNgSignal(
```

```
                false), mMitoticMode(0), mSeed(0), mp_PostMitoticType(), mp_RGC_Type(), mp_AC_HC_Type(), mp_PR_BC_Type(), mp_label_Type()
{
}

BoijeCellCycleModel::BoijeCellCycleModel(const BoijeCellCycleModel& rModel) :
        AbstractSimpleCellCycleModel(rModel), mOutput(rModel.mOutput), mEventStartTime(rModel.mEventStartTime), mSequenceSampler(
                rModel.mSequenceSampler), mSeqSamplerLabelSister(rModel.mSeqSamplerLabelSister), mDebug(rModel.mDebug), mTimeID(
                rModel.mTimeID), mVarIDs(rModel.mVarIDs), mDebugWriter(rModel.mDebugWriter), mGeneration(
                rModel.mGeneration), mPhase2gen(rModel.mPhase2gen), mPhase3gen(rModel.mPhase3gen), mprobAtoh7(
                rModel.mprobAtoh7), mprobPtf1a(rModel.mprobPtf1a), mprobng(rModel.mprobng), mAtoh7Signal(
                rModel.mAtoh7Signal), mPtf1aSignal(rModel.mPtf1aSignal), mNgSignal(rModel.mNgSignal), mMitoticMode(
                rModel.mMitoticMode), mSeed(rModel.mSeed), mp_PostMitoticType(rModel.mp_PostMitoticType), mp_RGC_Type(
                rModel.mp_RGC_Type), mp_AC_HC_Type(rModel.mp_AC_HC_Type), mp_PR_BC_Type(rModel.mp_PR_BC_Type), mp_label_Type(
                rModel.mp_label_Type)
{
}

AbstractCellCycleModel* BoijeCellCycleModel::CreateCellCycleModel()
{
    return new BoijeCellCycleModel(*this);
}

void BoijeCellCycleModel::SetCellCycleDuration()
{

    if (mpCell->GetCellProliferativeType()->IsType<DifferentiatedCellProliferativeTy
    {
        mCellCycleDuration = DBL_MAX;
    }
```

```
    else
    {
        mCellCycleDuration = 1.0;
    }

}

void BoijeCellCycleModel::ResetForDivision()
{
    mGeneration++; //increment generation counter
    //the first division is ascribed to generation "1"

    RandomNumberGenerator* p_random_number_generator = RandomNumberGenerator::Instance();

    mMitoticMode = 0; //0=PP;1=PD;2=DD

    /***********************************************
     * TRANSCRIPTION FACTOR RANDOM VARIABLES & RULES
     * 1st phase: only PP divisions (symmetric proliferative) permitted
     * 2nd phase: all division modes permitted, all TF signals avail
     * 3rd phase: only PP/DD modes permitted, only ng has nonzero signal
     ***********************************************/

    //reset RVs and signals
    double atoh7RV = 1, ptf1aRV = 1, ngRV = 1;
    mAtoh7Signal = mPtf1aSignal = mNgSignal = false;


    //PHASE & TF SIGNAL RULES
    if (mGeneration > mPhase2gen && mGeneration ≤ mPhase3gen) //if the cell is in the 2nd model phase, all signals have nonzero probabilities at each division
    {
        //RVs take values evenly distributed across 0-1
        atoh7RV = p_random_number_generator→ranf();
        ptf1aRV = p_random_number_generator→ranf();
        ngRV = p_random_number_generator→ranf();

        if (atoh7RV < mprobAtoh7)
        {
            mAtoh7Signal = true;
        }
```

```
        if (ptf1aRV < mprobPtf1a)
        {
            mPtf1aSignal = true;
        }
        if (ngRV < mprobng)
        {
            mNgSignal = true;
        }
    }

    if (mGeneration > mPhase3gen) //if the cell is in the 3rd model phase, only ng signal has a nonzero probability
    {
        ngRV = p_random_number_generator->ranf();
        //roll a probability die for the ng signal
        if (ngRV < mprobng)
        {
            mNgSignal = true;
        }
    }

    /****************
     * MITOTIC MODE & SPECIFICATION RULES
     * -(additional asymmetric postmitotic rules specified in InitialiseDaughterCell();)
     * *************/

    if(mAtoh7Signal == true)
    {
        mMitoticMode = 1;
    }

    if (mPtf1aSignal == true && mAtoh7Signal == false) //Ptf1A alone gives a symmetrical postmitotic AC/HC division
    {
        mMitoticMode = 2;
        mpCell->SetCellProliferativeType(mp_PostMitoticType);
        mpCell->AddCellProperty(mp_AC_HC_Type);
    }

    if (mPtf1aSignal == false && mAtoh7Signal == false && mNgSignal == true) //ng alone gives a symmetrical postmitotic PR/BC division
    {
```

```
        mMitoticMode = 2;
        mpCell->SetCellProliferativeType(mp_PostMitoticType);
        mpCell->AddCellProperty(mp_PR_BC_Type);
    }

    /****************
     * Write mitotic event to file if appropriate
     * mDebug: many files: detailed per-lineage info switch; intended for TestHeInductionCountFixture
     * mOutput: 1 file: time, seed, cellID, mitotic mode, intended for TestHeMitoticModeRateFixture
     * *************/

    if (mDebug)
    {
        WriteDebugData(atoh7RV, ptf1aRV, ngRV);
    }

    if (mOutput)
    {
        WriteModeEventOutput();
    }

    AbstractSimpleCellCycleModel::ResetForDivision();

    /******************
     * SEQUENCE SAMPLER
     ******************/
    //if the sequence sampler has been turned on, check for the label & write mitotic mode to log
    //50% chance of each daughter cell from a mitosis inheriting the label
    if (mSequenceSampler)
    {
        if (mpCell->HasCellProperty<CellLabel>())
        {
            (*LogFile::Instance()) << mMitoticMode;
            double labelRV = p_random_number_generator->ranf();
            if (labelRV <= .5)
            {
                mSeqSamplerLabelSister = true;
                mpCell->RemoveCellProperty<CellLabel>();
            }
```

```
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         {
```

```
            mpCell->RemoveCellProperty<CellLabel>();
        }
    }
}

void BoijeCellCycleModel::SetGeneration(unsigned generation)
{
    mGeneration = generation;
}

unsigned BoijeCellCycleModel::GetGeneration() const
{
    return mGeneration;
}

void BoijeCellCycleModel::SetPostMitoticType(boost::shared_ptr<AbstractCellProperty> p_PostMitoticType)
{
    mp_PostMitoticType = p_PostMitoticType;
}
void BoijeCellCycleModel::SetSpecifiedTypes(boost::shared_ptr<AbstractCellProperty> p_RGC_Type,
                                            boost::shared_ptr<AbstractCellPropert p_AC_HC_Type,
                                            boost::shared_ptr<AbstractCellPropert p_PR_BC_Type)
{
    mp_RGC_Type = p_RGC_Type;
    mp_AC_HC_Type = p_AC_HC_Type;
    mp_PR_BC_Type = p_PR_BC_Type;
}

void BoijeCellCycleModel::SetModelParameters(unsigned phase2gen, unsigned phase3gen, double probAtoh7, double probPtf1a,
                                             double probng)
{
    mPhase2gen = phase2gen;
    mPhase3gen = phase3gen;
    mprobAtoh7 = probAtoh7;
```

```
    mprobPtf1a = probPtf1a;
    mprobng = probng;
}

void BoijeCellCycleModel::EnableModeEventOutput(double eventStart, unsigned seed)
{
    mOutput = true;
    mEventStartTime = eventStart;
    mSeed = seed;

}

void BoijeCellCycleModel::WriteModeEventOutput()
{
    double currentTime = SimulationTime::Instance()->GetTime() + mEventStartTime;
    CellPtr currentCell = GetCell();
    double currentCellID = (double) currentCell->GetCellId();
    (*LogFile::Instance()) << currentTime << "\t" << mSeed << "\t" << currentCellID << "\t" << mMitoticMode << "\n";
}

void BoijeCellCycleModel::EnableSequenceSampler(boost::shared_ptr<AbstractCellProperty label)
{
    mSequenceSampler = true;
    mp_label_Type = label;
}

void BoijeCellCycleModel::EnableModelDebugOutput(boost::shared_ptr<ColumnDataWriter> debugWriter)
{
    mDebug = true;
    mDebugWriter = debugWriter;

    mTimeID = mDebugWriter->DefineUnlimitedDimension("Time", "h");
    mVarIDs.push_back(mDebugWriter->DefineVariable("CellID", "No"));
    mVarIDs.push_back(mDebugWriter->DefineVariable("Generation", "No"));
    mVarIDs.push_back(mDebugWriter->DefineVariable("MitoticMode", "Mode"));
```

```
    mVarIDs.push_back(mDebugWriter->DefineVariable("atoh7Set", "Percentile"));
    mVarIDs.push_back(mDebugWriter->DefineVariable("atoh7RV", "Percentile"));
    mVarIDs.push_back(mDebugWriter->DefineVariable("ptf1aSet", "Percentile"));
    mVarIDs.push_back(mDebugWriter->DefineVariable("ptf1aRV", "Percentile"));
    mVarIDs.push_back(mDebugWriter->DefineVariable("ngSet", "Percentile"));
    mVarIDs.push_back(mDebugWriter->DefineVariable("ngRV", "Percentile"));

    mDebugWriter->EndDefineMode();
}

void BoijeCellCycleModel::WriteDebugData(double atoh7RV, double ptf1aRV, double ngRV)
{
    double currentTime = SimulationTime::Instance()->GetTime();
    CellPtr currentCell = GetCell();
    double currentCellID = (double) currentCell->GetCellId();

    mDebugWriter->PutVariable(mTimeID, currentTime);
    mDebugWriter->PutVariable(mVarIDs[0], currentCellID);
    mDebugWriter->PutVariable(mVarIDs[1], mGeneration);
    mDebugWriter->PutVariable(mVarIDs[2], mMitoticMode);
    mDebugWriter->PutVariable(mVarIDs[3], mprobAtoh7);
    mDebugWriter->PutVariable(mVarIDs[4], atoh7RV);
    mDebugWriter->PutVariable(mVarIDs[5], mprobPtf1a);
    mDebugWriter->PutVariable(mVarIDs[6], ptf1aRV);
    mDebugWriter->PutVariable(mVarIDs[7], mprobng);
    mDebugWriter->PutVariable(mVarIDs[8], ngRV);
    mDebugWriter->AdvanceAlongUnlimitedDimension();
}

/******************
 * UNUSED FUNCTIONS (required for class nonvirtuality, do not remove)
 ******************/

double BoijeCellCycleModel::GetAverageTransitCellCycleTime()
{
    return mCellCycleDuration;
```

```
}

double BoijeCellCycleModel::GetAverageStemCellCycleTime()
{
    return mCellCycleDuration;
}

void BoijeCellCycleModel::OutputCellCycleModelParameters(out_stream&
 ↪  rParamsFile)
{
    *rParamsFile << "\t\t\t<CellCycleDuration>" << mCellCycleDuration <<
     ↪  "</CellCycleDuration>\n";

    // Call method on direct parent class

     ↪  AbstractSimpleCellCycleModel::OutputCellCycleModelParameters(rParamsFile);
}

// Serialization for Boost ≥ 1.36
#include "SerializationExportWrapperForCpp.hpp"
CHASTE_CLASS_EXPORT(BoijeCellCycleModel)
```

15.1.18 /src/BoijeCellCycleModel.hpp

```
#ifndef BOIJECELLCYCLEMODEL_HPP_
#define BOIJECELLCYCLEMODEL_HPP_

#include "AbstractSimpleCellCycleModel.hpp"
#include "RandomNumberGenerator.hpp"
#include "Cell.hpp"
#include "DifferentiatedCellProliferativeType.hpp"
#include "SmartPointers.hpp"
#include "ColumnDataWriter.hpp"
#include "LogFile.hpp"
#include "CellLabel.hpp"

#include "BoijeRetinalNeuralFates.hpp"


/******************************
* 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
```

```
     * @param version the current version of this class
     */
    template<class Archive>
    void serialize(Archive & archive, const unsigned int version)
    {
        archive & boost::serialization::base_object<AbstractSimpleCellCycleModel>(*this);

        SerializableSingleton<RandomNumberGenerator>* p_wrapper = RandomNumberGenerator::Instance()->GetSerializationWrapper();
        archive & p_wrapper;
        archive & mCellCycleDuration;
        archive & mGeneration;
    }

    //Private write functions for models
    void WriteModeEventOutput();
    void WriteDebugData(double atoh7RV, double ptf1aRV, double ngRV);

protected:
    //mode/output variables
    bool mOutput;
    double mEventStartTime;
    bool mSequenceSampler;
    bool mSeqSamplerLabelSister;
    //debug writer stuff
    bool mDebug;
    int mTimeID;
    std::vector<int> mVarIDs;
    boost::shared_ptr<ColumnDataWriter> mDebugWriter;
    //model parameters and state memory vars
    unsigned mGeneration;
    unsigned mPhase2gen;
    unsigned mPhase3gen;
    double mprobAtoh7;
        double mprobPtf1a;
        double mprobng;
        bool mAtoh7Signal;
        bool mPtf1aSignal;
        bool mNgSignal;
    unsigned mMitoticMode;
    unsigned mSeed;
    boost::shared_ptr<AbstractCellProperty> mp_PostMitoticType;
```

```
    boost::shared_ptr<AbstractCellProperty> mp_RGC_Type;
    boost::shared_ptr<AbstractCellProperty> mp_AC_HC_Type;
    boost::shared_ptr<AbstractCellProperty> mp_PR_BC_Type;
    boost::shared_ptr<AbstractCellProperty> mp_label_Type;

    /**
     * Protected copy-constructor for use by CreateCellCycleModel().
     *
     * The only way for external code to create a copy of a cell cycle model
     * is by calling that method, to ensure that a model of the correct subclass is created.
     * This copy-constructor helps subclasses to ensure that all member variables are correctly copied when this happens.
     *
     * This method is called by child classes to set member variables for a daughter cell upon cell division.
     * Note that the parent cell cycle model will have had ResetForDivision() called just before CreateCellCycleModel() is called,
     * so performing an exact copy of the parent is suitable behaviour. Any daughter-cell-specific initialisation
     * can be done in InitialiseDaughterCell().
     *
     * @param rModel the cell cycle model to copy.
     */
    BoijeCellCycleModel(const BoijeCellCycleModel& rModel);

public:

    /**
     * Constructor - just a default, mBirthTime is set in the AbstractCellCycleModel class.
     */
    BoijeCellCycleModel();

    /**
     * SetCellCycleDuration() method to set length of cell cycle (default 1.0 as Boije's model is purely generational)
     */
    void SetCellCycleDuration();

    /**
```

```
     * Overridden builder method to create new copies of
     * this cell-cycle model.
     *
     * @return new cell-cycle model
     */
    AbstractCellCycleModel* CreateCellCycleModel();

    /**
     * Overridden ResetForDivision() method.
     * Contains general mitotic mode logic
     **/
    void ResetForDivision();

    /**
     * Overridden InitialiseDaughterCell() method.
     * Used to implement asymmetric mitotic mode
     * */
    void InitialiseDaughterCell();

    /**
     * Sets the cell's generation.
     *
     * @param generation the cell's generation
     **/
    void SetGeneration(unsigned generation);

    /**
     * @return the cell's generation.
     */
    unsigned GetGeneration() const;

    /****************************
     * Model setup functions:
     * Set TF signal probabilities with SetModelParameters
     * Set Postmitotic and specification AbstractCellProperties w/ the
↪   relevant functions
     * By default these are available in BoijeRetinalNeuralFates.hpp
     *
     * NB: The Boije model lumps together amacrine & horizontal cells,
↪   photoreceptors & bipolar neurons
     ****************************/

```

```
    void SetModelParameters(unsigned phase2gen = 3, unsigned phase3gen = 5, double probAtoh7 = 0.32, double probPtf1a = 0.3, double probng = 0.8);
    void SetPostMitoticType(boost::shared_ptr<AbstractCellProperty> p_PostMitoticType);
    void SetSpecifiedTypes(boost::shared_ptr<AbstractCellProperty> p_RGC_Type, boost::shared_ptr<AbstractCellProperty> p_AC_HC_Type, boost::shared_ptr<AbstractCellProperty> p_PR_BC_Type);

    //Functions to enable per-cell mitotic mode logging for mode rate & sequence sampling fixtures
    //Uses singleton logfile
    void EnableModeEventOutput(double eventStart, unsigned seed);
    void EnableSequenceSampler(boost::shared_ptr<AbstractCellProperty> label);

    //More detailed debug output. Needs a ColumnDataWriter passed to it
    //Only declare ColumnDataWriter directory, filename, etc; do not set up otherwise
    void EnableModelDebugOutput(boost::shared_ptr<ColumnDataWriter> debugWriter);

    /**
     * Overridden GetAverageTransitCellCycleTime() method.
     *
     * @return the average of mMinCellCycleDuration and mMaxCellCycleDuration
     */
    double GetAverageTransitCellCycleTime();

    /**
     * Overridden GetAverageStemCellCycleTime() method.
     *
     * @return the average of mMinCellCycleDuration and mMaxCellCycleDuration
     */
    double GetAverageStemCellCycleTime();

    /**
     * Overridden OutputCellCycleModelParameters() method.
     *
     * @param rParamsFile the file stream to which the parameters are output
```

```
     */
    virtual void OutputCellCycleModelParameters(out_stream& rParamsFile);
};

#include "SerializationExportWrapper.hpp"
// Declare identifier for the serializer
CHASTE_CLASS_EXPORT(BoijeCellCycleModel)

#endif /*BOIJECELLCYCLEMODEL_HPP_*/
```

15.1.19 /src/BoijeRetinalNeuralFates.cpp

```
#include "../../../../projects/ISP/src/BoijeRetinalNeuralFates.hpp"

RetinalGanglion::RetinalGanglion(unsigned colour)
    : AbstractCellProperty(),
      mColour(colour)
{
}

RetinalGanglion::~RetinalGanglion()
{
}

unsigned RetinalGanglion::GetColour() const
{
    return mColour;
}

AmacrineHorizontal::AmacrineHorizontal(unsigned colour)
    : AbstractCellProperty(),
      mColour(colour)
{
}

AmacrineHorizontal::~AmacrineHorizontal()
{
}

unsigned AmacrineHorizontal::GetColour() const
{
    return mColour;
```

```
}

ReceptorBipolar::ReceptorBipolar(unsigned colour)
    : AbstractCellProperty(),
      mColour(colour)
{
}

ReceptorBipolar::~ReceptorBipolar()
{
}

unsigned ReceptorBipolar::GetColour() const
{
    return mColour;
}

#include "SerializationExportWrapperForCpp.hpp"
// Declare identifier for the serializer
CHASTE_CLASS_EXPORT(RetinalGanglion)
CHASTE_CLASS_EXPORT(AmacrineHorizontal)
CHASTE_CLASS_EXPORT(ReceptorBipolar)
```

15.1.20 /src/BoijeRetinalNeuralFates.hpp

```
#ifndef BOIJERETINALNEURALFATES_HPP_
#define BOIJERETINALNEURALFATES_HPP_

#include <boost/shared_ptr.hpp>
#include "AbstractCellProperty.hpp"
#include "ChasteSerialization.hpp"
#include <boost/serialization/base_object.hpp>

class RetinalGanglion : public AbstractCellProperty
{
private:

    /**
     * Colour for use by visualizer.
     */
    unsigned mColour;

```

```
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 &
            ↪  boost::serialization::base_object<AbstractCellProperty>(*this);
30          archive & mColour;
31      }
32
33  public:
34
35      /**
36       * Constructor.
37       *
38       * @param colour  what colour cells with this property should be in
    ↪   the visualizer (defaults to 6)
39       */
40      RetinalGanglion(unsigned colour=3);
41
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      /**
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 &
           ↪  boost::serialization::base_object<AbstractCellProperty>(*this);
74         archive & mColour;
75     }
76
77 public:
78
79     /**
80      * Constructor.
81      *
82      * @param colour  what colour cells with this property should be in
   ↪  the visualizer (defaults to 6)
83      */
84     AmacrineHorizontal(unsigned colour=4);
85
86     /**
87      * Destructor.
88      */
89     virtual ~AmacrineHorizontal();
90
91     /**
92      * @return #mColour.
93      */
94     unsigned GetColour() const;
95 };
96
97 class ReceptorBipolar : public AbstractCellProperty
98 {
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 &
            ↪  boost::serialization::base_object<AbstractCellProperty>(*this);
118         archive & mColour;
119     }
120
121 public:
122
123     /**
124      * Constructor.
125      *
126      * @param colour  what colour cells with this property should be in
    ↪  the visualizer (defaults to 6)
127      */
128     ReceptorBipolar(unsigned colour=5);
129
130     /**
131      * Destructor.
132      */
133     virtual ~ReceptorBipolar();
134
135     /**
136      * @return #mColour.
137      */
138     unsigned GetColour() const;
139 };
140
```

```
#include "SerializationExportWrapper.hpp"
// Declare identifier for the serializer
CHASTE_CLASS_EXPORT(RetinalGanglion)
CHASTE_CLASS_EXPORT(AmacrineHorizontal)
CHASTE_CLASS_EXPORT(ReceptorBipolar)

#endif /* BOIJERETINALNEURALFATES_HPP_ */
```

15.1.21 /src/GomesCellCycleModel.cpp

```
#include "GomesCellCycleModel.hpp"
#include "GomesRetinalNeuralFates.hpp"

GomesCellCycleModel::GomesCellCycleModel() :
        AbstractSimpleCellCycleModel(), mOutput(false), mEventStartTime(), mSequenceSampler(false), mSeqSamplerLabelSister(
                false), mDebug(false), mTimeID(), mVarIDs(), mDebugWriter(), mNormalMu(3.9716), mNormalSigma(0.32839), mPP(
                .055), mPD(0.221), mpBC(.128), mpAC(.106), mpMG(.028), mMitoticMode(), mSeed(), mp_PostMitoticType(), mp_RPh_Type(), mp_BC_Type(), mp_AC_Type(), mp_MG_Type(), mp_label_Type()
{
}

GomesCellCycleModel::GomesCellCycleModel(const GomesCellCycleModel& rModel) :
        AbstractSimpleCellCycleModel(rModel), mOutput(rModel.mOutput), mEventStartTime(rModel.mEventStartTime), mSequenceSampler(
                rModel.mSequenceSampler), mSeqSamplerLabelSister(rModel.mSeqSamplerLabelSister), mDebug(rModel.mDebug), mTimeID(
                rModel.mTimeID), mVarIDs(rModel.mVarIDs), mDebugWriter(rModel.mDebugWriter), mNormalMu(
                rModel.mNormalMu), mNormalSigma(rModel.mNormalSigma), mPP(rModel.mPP), mPD(rModel.mPD), mpBC(
                rModel.mpBC), mpAC(rModel.mpAC), mpMG(rModel.mpMG), mMitoticMode(rModel.mMitoticMode), mSeed(
```

```
                rModel.mSeed),
                ↪ mp_PostMitoticType(rModel.mp_PostMitoticType),
                ↪ mp_RPh_Type(rModel.mp_RPh_Type), mp_BC_Type(
                rModel.mp_BC_Type), mp_AC_Type(rModel.mp_AC_Type),
                ↪ mp_MG_Type(rModel.mp_MG_Type), mp_label_Type(
                rModel.mp_label_Type)
{
}

AbstractCellCycleModel* GomesCellCycleModel::CreateCellCycleModel()
{
    return new GomesCellCycleModel(*this);
}

void GomesCellCycleModel::SetCellCycleDuration()
{
    /*************************************
     * CELL CYCLE DURATION RANDOM VARIABLE
     *************************************/

    RandomNumberGenerator* p_random_number_generator =
    ↪ RandomNumberGenerator::Instance();

    //Gomes cell cycle length determined by lognormal distribution with
    ↪ default mean 56 hr, std 18.9 hrs.
    mCellCycleDuration =
    ↪ exp(p_random_number_generator->NormalRandomDeviate(mNormalMu,
    ↪ mNormalSigma));
}

void GomesCellCycleModel::ResetForDivision()
{
    /****************
     * Mitotic mode rules
     * *************/
    RandomNumberGenerator* p_random_number_generator =
    ↪ RandomNumberGenerator::Instance();

    //Check time in lineage and determine current mitotic mode phase
    mMitoticMode = 0; //0=PP;1=PD;2=DD

    /******************************
     * MITOTIC MODE RANDOM VARIABLE
```

```
     ******************************/
    //initialise mitoticmode random variable, set mitotic mode appropriately after comparing to mode probability array
    double mitoticModeRV = p_random_number_generator->ranf();

    if (mitoticModeRV > mPP && mitoticModeRV <= mPP + mPD)
    {
        mMitoticMode = 1;
    }
    if (mitoticModeRV > mPP + mPD)
    {
        mMitoticMode = 2;
    }

    /****************
     * Write mitotic event to file if appropriate
     * mDebug: many files: detailed per-lineage info switch; intended for TestHeInductionCountFixture
     * mOutput: 1 file: time, seed, cellID, mitotic mode, intended for TestHeMitoticModeRateFixture
     * *************/

    if (mDebug)
    {
        WriteDebugData(mitoticModeRV);
    }

    if (mOutput)
    {
        WriteModeEventOutput();
    }

    //set new cell cycle length (will be overwritten with DBL_MAX for DD divisions)
    AbstractSimpleCellCycleModel::ResetForDivision();

    /****************
     * Symmetric postmitotic specification rule
     * -(asymmetric postmitotic rule specified in InitialiseDaughterCell();)
     * *************/

    if (mMitoticMode == 2)
```

```
    {
        mpCell->SetCellProliferativeType(mp_PostMitoticType);
        mCellCycleDuration = DBL_MAX;
        /****************************
         * SPECIFICATION RANDOM VARIABLE
         ****************************/
        double specificationRV = p_random_number_generator->ranf();
        if (specificationRV <= mpMG)
        {
            mpCell->AddCellProperty(mp_MG_Type);
        }
        if (specificationRV > mpMG && specificationRV <= mpMG + mpAC)
        {
            mpCell->AddCellProperty(mp_AC_Type);
        }
        if (specificationRV > mpMG + mpAC && specificationRV <= mpMG +
        ↪  mpAC + mpBC)
        {
            mpCell->AddCellProperty(mp_BC_Type);
        }
        if (specificationRV > mpMG + mpAC + mpBC)
        {
            mpCell->AddCellProperty(mp_RPh_Type);
        }
    }

    /******************
     * SEQUENCE SAMPLER
     ******************/
    //if the sequence sampler has been turned on, check for the label &
    ↪  write mitotic mode to log
    //50% chance of each daughter cell from a mitosis inheriting the
    ↪  label
    if (mSequenceSampler)
    {
        if (mpCell->HasCellProperty<CellLabel>())
        {
            (*LogFile::Instance()) << mMitoticMode;
            double labelRV = p_random_number_generator->ranf();
            if (labelRV <= .5)
            {
                mSeqSamplerLabelSister = true;
                mpCell->RemoveCellProperty<CellLabel>();
```

```
            }
            else
            {
                mSeqSamplerLabelSister = false;
            }
        }
        else
        {
            //prevents lost-label cells from labelling their progeny
            mSeqSamplerLabelSister = false;
        }
    }
}

void GomesCellCycleModel::InitialiseDaughterCell()
{
    if (mMitoticMode == 0)
    {
        //daughter cell's mCellCycleDuration is copied from parent; reset to new value from gamma PDF here
        SetCellCycleDuration();
    }

    /************
     * PD-type division
     **********/

    if (mMitoticMode == 1)
    {
        RandomNumberGenerator* p_random_number_generator = RandomNumberGenerator::Instance();
        mpCell->SetCellProliferativeType(mp_PostMitoticType);
        mCellCycleDuration = DBL_MAX;
        /********************
         * SPECIFICATION RULES
         *******************/
        double specificationRV = p_random_number_generator->ranf();
        if (specificationRV <= mpMG)
        {
            mpCell->AddCellProperty(mp_MG_Type);
        }
        if (specificationRV > mpMG && specificationRV <= mpMG + mpAC)
        {
```

```
            mpCell->AddCellProperty(mp_AC_Type);
        }
        if (specificationRV > mpMG + mpAC && specificationRV ≤ mpMG + mpAC + mpBC)
        {
            mpCell->AddCellProperty(mp_BC_Type);
        }
        if (specificationRV > mpMG + mpAC + mpBC)
        {
            mpCell->AddCellProperty(mp_RPh_Type);
        }
    }

    if (mMitoticMode == 2)
    {
        RandomNumberGenerator* p_random_number_generator = RandomNumberGenerator::Instance();
        //remove the fate assigned to the parent cell in ResetForDivision, then assign the sister fate as usual
        mpCell->RemoveCellProperty<AbstractCellProperty>();
        mpCell->SetCellProliferativeType(mp_PostMitoticType);

        /*********************
         * SPECIFICATION RULES
         ********************/
        double specificationRV = p_random_number_generator->ranf();
        if (specificationRV ≤ mpMG)
        {
            mpCell->AddCellProperty(mp_MG_Type);
        }
        if (specificationRV > mpMG && specificationRV ≤ mpMG + mpAC)
        {
            mpCell->AddCellProperty(mp_AC_Type);
        }
        if (specificationRV > mpMG + mpAC && specificationRV ≤ mpMG + mpAC + mpBC)
        {
            mpCell->AddCellProperty(mp_BC_Type);
        }
        if (specificationRV > mpMG + mpAC + mpBC)
        {
            mpCell->AddCellProperty(mp_RPh_Type);
        }
```

```
    }

    /******************
     * SEQUENCE SAMPLER
     ******************/
    if (mSequenceSampler)
    {
        if (mSeqSamplerLabelSister)
        {
            mpCell->AddCellProperty(mp_label_Type);
            mSeqSamplerLabelSister = false;
        }
        else
        {
            mpCell->RemoveCellProperty<CellLabel>();
        }
    }
}

void GomesCellCycleModel::SetModelParameters(const double normalMu, const double normalSigma, const double PP,
                                             const double PD, const double pBC, const double pAC, const double pMG)
{
    mNormalMu = normalMu;
    mNormalSigma = normalSigma;
    mPP = PP;
    mPD = PD;
    mpBC = pBC;
    mpAC = pAC;
    mpMG = pMG;

}

void GomesCellCycleModel::SetModelProperties(boost::shared_ptr<AbstractCellProperty> p_RPh_Type,

                                             boost::shared_ptr<AbstractCellProper
                                             p_AC_Type,
```

```
                                                    ↪  boost::shared_ptr<AbstractCellProper
                                                    ↪  p_BC_Type,

                                                    ↪  boost::shared_ptr<AbstractCellProper
                                                    ↪  p_MG_Type)
{
    mp_RPh_Type = p_RPh_Type;
    mp_AC_Type = p_AC_Type;
    mp_BC_Type = p_BC_Type;
    mp_MG_Type = p_MG_Type;
}

void
 ↪  GomesCellCycleModel::SetPostMitoticType(boost::shared_ptr<AbstractCellProperty>
 ↪  p_PostMitoticType)
{
    mp_PostMitoticType = p_PostMitoticType;
}

void GomesCellCycleModel::EnableModeEventOutput(double eventStart,
 ↪  unsigned seed)
{
    mOutput = true;
    mEventStartTime = eventStart;
    mSeed = seed;

}

void GomesCellCycleModel::WriteModeEventOutput()
{
    double currentTime = SimulationTime::Instance()->GetTime() +
     ↪  mEventStartTime;
    CellPtr currentCell = GetCell();
    double currentCellID = (double) currentCell->GetCellId();
    (*LogFile::Instance()) << currentTime << "\t" << mSeed << "\t" <<
     ↪  currentCellID << "\t" << mMitoticMode << "\n";
}

void
 ↪  GomesCellCycleModel::EnableSequenceSampler(boost::shared_ptr<AbstractCellProperty
 ↪  label)
{
```

```
    mSequenceSampler = true;
    mp_label_Type = label;
}

void
↪ GomesCellCycleModel::EnableModelDebugOutput(boost::shared_ptr<ColumnDataWriter>
↪ debugWriter)
{
    mDebug = true;
    mDebugWriter = debugWriter;

    mTimeID = mDebugWriter->DefineUnlimitedDimension("Time", "h");
    mVarIDs.push_back(mDebugWriter->DefineVariable("CellID", "No"));
    mVarIDs.push_back(mDebugWriter->DefineVariable("CycleDuration",
     ↪ "h"));
    mVarIDs.push_back(mDebugWriter->DefineVariable("PP", "Percentile"));
    mVarIDs.push_back(mDebugWriter->DefineVariable("PD", "Percentile"));
    mVarIDs.push_back(mDebugWriter->DefineVariable("Dieroll",
     ↪ "Percentile"));
    mVarIDs.push_back(mDebugWriter->DefineVariable("MitoticMode",
     ↪ "Mode"));

    mDebugWriter->EndDefineMode();
}

void GomesCellCycleModel::WriteDebugData(double percentileRoll)
{
    double currentTime = SimulationTime::Instance()->GetTime();
    CellPtr currentCell = GetCell();
    double currentCellID = (double) currentCell->GetCellId();

    mDebugWriter->PutVariable(mTimeID, currentTime);
    mDebugWriter->PutVariable(mVarIDs[0], currentCellID);
    mDebugWriter->PutVariable(mVarIDs[1], mCellCycleDuration);
    mDebugWriter->PutVariable(mVarIDs[2], mPP);
    mDebugWriter->PutVariable(mVarIDs[3], mPD);
    mDebugWriter->PutVariable(mVarIDs[4], percentileRoll);
    mDebugWriter->PutVariable(mVarIDs[5], mMitoticMode);
    mDebugWriter->AdvanceAlongUnlimitedDimension();
}

/******************
 * UNUSED FUNCTIONS (required for class nonvirtuality, do not remove)
```

```
 ******************/

double GomesCellCycleModel::GetAverageTransitCellCycleTime()
{
    return (0.0);
}

double GomesCellCycleModel::GetAverageStemCellCycleTime()
{
    return (0.0);
}

void GomesCellCycleModel::OutputCellCycleModelParameters(out_stream&
 ↪  rParamsFile)
{
    *rParamsFile << "\t\t\t<CellCycleDuration>" << mCellCycleDuration <<
     ↪  "</CellCycleDuration>\n";

    // Call method on direct parent class

     ↪  AbstractSimpleCellCycleModel::OutputCellCycleModelParameters(rParamsFile);
}

// Serialization for Boost ≥ 1.36
#include "SerializationExportWrapperForCpp.hpp"
CHASTE_CLASS_EXPORT(GomesCellCycleModel)
```

15.1.22 /src/GomesCellCycleModel.hpp

```
#ifndef GOMESCELLCYCLEMODEL_HPP_
#define GOMESCELLCYCLEMODEL_HPP_

#include "AbstractSimpleCellCycleModel.hpp"
#include "RandomNumberGenerator.hpp"
#include "Cell.hpp"
#include "DifferentiatedCellProliferativeType.hpp"
#include "GomesRetinalNeuralFates.hpp"
#include "SmartPointers.hpp"
#include "ColumnDataWriter.hpp"
#include "LogFile.hpp"
#include "CellLabel.hpp"

```

```
/*****************************************
 * GOMES CELL CYCLE MODEL
 * As described in Gomes et al. 2011 [Gomes2011] doi: 10.1242/dev.059683
 *
 * USE: By default, GomesCellCycleModels are constructed with the parameter fit reported in [Gomes2011].
 * Cell cycle length, mitotic mode, and postmitotic fate of cells are determined by independent random variables
 * PP = symmetric proliferative mitotic mode, both progeny remain mitotic
 * PD = asymmetric proliferative mitotic mode, one progeny exits the cell cycle and differentiates
 * DD = symmetric differentiative mitotic mode, both progeny exit the cell cycle and differentiate
 *
 * Change default model parameters with SetModelParameters(<params>);
 * Set AbstractCellProperties for differentiated neural types with SetModelProperties();
 *
 * 2 per-model-event output modes:
 * EnableModeEventOutput() enables mitotic mode event logging-all cells will write to the singleton log file
 * EnableModelDebugOutput() enables more detailed debug output, each seed will have its own file written to
 * by a ColumnDataWriter passed to it from the test
 * (eg. by the SetupDebugOutput helper function in the project simulator)
 *
 * 1 mitotic-event-sequence sampler (only samples one "path" through the lineage):
 * EnableSequenceSampler() - one "sequence" of progenitors writes mitotic event type to a string in the singleton log file
 *
 *********************************************/

class GomesCellCycleModel : public AbstractSimpleCellCycleModel
{
    friend class TestSimpleCellCycleModels;

private:

    /** Needed for serialization. */
    friend class boost::serialization::access;
    /**
```

```
     * Archive the cell-cycle model and random number generator, never used directly - boost uses this.
     *
     * @param archive the archive
     * @param version the current version of this class
     */
    template<class Archive>
    void serialize(Archive & archive, const unsigned int version)
    {
        archive & boost::serialization::base_object<AbstractSimpleCellCycleModel>(*this);

        SerializableSingleton<RandomNumberGenerator>* p_wrapper = RandomNumberGenerator::Instance()->GetSerializationWrapper();
        archive & p_wrapper;
        archive & mCellCycleDuration;
    }

    //Private write functions for models
    void WriteModeEventOutput();
    void WriteDebugData(double percentile);

protected:
    //mode/output variables
    bool mOutput;
    double mEventStartTime;
    bool mSequenceSampler;
    bool mSeqSamplerLabelSister;
    //debug writer stuff
    bool mDebug;
    int mTimeID;
    std::vector<int> mVarIDs;
    boost::shared_ptr<ColumnDataWriter> mDebugWriter;
    //model parameters and state memory vars
    double mNormalMu;
    double mNormalSigma;
    double mPP;
    double mPD;
    double mpBC;
    double mpAC;
    double mpMG;
    unsigned mMitoticMode;
```

```
    unsigned mSeed;
    boost::shared_ptr<AbstractCellProperty> mp_PostMitoticType;
    boost::shared_ptr<AbstractCellProperty> mp_RPh_Type;
    boost::shared_ptr<AbstractCellProperty> mp_BC_Type;
    boost::shared_ptr<AbstractCellProperty> mp_AC_Type;
    boost::shared_ptr<AbstractCellProperty> mp_MG_Type;
    boost::shared_ptr<AbstractCellProperty> mp_label_Type;

    /**
     * Protected copy-constructor for use by CreateCellCycleModel().
     *
     * The only way for external code to create a copy of a cell cycle model
     * is by calling that method, to ensure that a model of the correct subclass is created.
     * This copy-constructor helps subclasses to ensure that all member variables are correctly copied when this happens.
     *
     * This method is called by child classes to set member variables for a daughter cell upon cell division.
     * Note that the parent cell cycle model will have had ResetForDivision() called just before CreateCellCycleModel() is called,
     * so performing an exact copy of the parent is suitable behaviour. Any daughter-cell-specific initialisation
     * can be done in InitialiseDaughterCell().
     *
     * @param rModel the cell cycle model to copy.
     */
    GomesCellCycleModel(const GomesCellCycleModel& rModel);

public:

    /**
     * Constructor - just a default, mBirthTime is set in the AbstractCellCycleModel class.
     */
    GomesCellCycleModel();

    /**
     * SetCellCycleDuration() method to set length of cell cycle (lognormal distribution as specified in [Gomes2011])
     */
```

```
    void SetCellCycleDuration();

    /**
     * Overridden builder method to create new copies of
     * this cell-cycle model.
     *
     * @return new cell-cycle model
     */
    AbstractCellCycleModel* CreateCellCycleModel();

    /**
     * Overridden ResetForDivision() method.
     * Contains general mitotic mode logic
     **/
    void ResetForDivision();

    /** Overridden InitialiseDaughterCell() method. Used to implement asymmetric mitotic mode*/
    void InitialiseDaughterCell();

    /* Model setup functions
     * Set lognormal cell cycle curve properties with *mean and std of corresponding NORMAL curve*
     * Model requires valid AbstractCellProperties to assign postmitotic fate;
     * Defaults are found in GomesRetinalNeuralFates.hpp
     * (RodPohotreceptor, AmacrineCell, BipolarCell, MullerGlia)
     */

    void SetModelParameters(const double normalMu = 3.9716, const double normalSigma = .32839, const double PP = .055,
                            const double PD = .221, const double pBC = .128, const double pAC = .106, const double pMG =
                                .028);
    void SetModelProperties(boost::shared_ptr<AbstractCellProperty> p_RPh_Type,
                            boost::shared_ptr<AbstractCellProperty> p_AC_Type,
                            boost::shared_ptr<AbstractCellProperty> p_BC_Type,
                            boost::shared_ptr<AbstractCellProperty> p_MG_Type);
```

```

    //This should normally be a DifferentiatedCellProliferativeType
    void SetPostMitoticType(boost::shared_ptr<AbstractCellProperty> p_PostMitoticType);

    //Functions to enable per-cell mitotic mode logging for mode rate & sequence sampling fixtures
    //Uses singleton logfile
    void EnableModeEventOutput(double eventStart, unsigned seed);
    void EnableSequenceSampler(boost::shared_ptr<AbstractCellProperty> label);

    //More detailed debug output. Needs a ColumnDataWriter passed to it
    //Only declare ColumnDataWriter directory, filename, etc; do not set up otherwise
    void EnableModelDebugOutput(boost::shared_ptr<ColumnDataWriter> debugWriter);

    //Not used, but must be overwritten lest GomesCellCycleModels be abstract
    double GetAverageTransitCellCycleTime();
    double GetAverageStemCellCycleTime();

    /**
     * Overridden OutputCellCycleModelParameters() method.
     *
     * @param rParamsFile the file stream to which the parameters are output
     */
    virtual void OutputCellCycleModelParameters(out_stream& rParamsFile);
};

#include "SerializationExportWrapper.hpp"
// Declare identifier for the serializer
CHASTE_CLASS_EXPORT(GomesCellCycleModel)

#endif /*GOMESCELLCYCLEMODEL_HPP_*/
```

15.1.23 /src/GomesRetinalNeuralFates.cpp

```
#include "GomesRetinalNeuralFates.hpp"

```

```
RodPhotoreceptor::RodPhotoreceptor(unsigned colour)
    : AbstractCellProperty(),
      mColour(colour)
{
}

RodPhotoreceptor::~RodPhotoreceptor()
{
}

unsigned RodPhotoreceptor::GetColour() const
{
    return mColour;
}

AmacrineCell::AmacrineCell(unsigned colour)
    : AbstractCellProperty(),
      mColour(colour)
{
}

AmacrineCell::~AmacrineCell()
{
}

unsigned AmacrineCell::GetColour() const
{
    return mColour;
}

BipolarCell::BipolarCell(unsigned colour)
    : AbstractCellProperty(),
      mColour(colour)
{
}

BipolarCell::~BipolarCell()
{
}

unsigned BipolarCell::GetColour() const
{
    return mColour;
```

```
}

MullerGlia::MullerGlia(unsigned colour)
    : AbstractCellProperty(),
      mColour(colour)
{
}

MullerGlia::~MullerGlia()
{
}

unsigned MullerGlia::GetColour() const
{
    return mColour;
}

#include "SerializationExportWrapperForCpp.hpp"
// Declare identifier for the serializer
CHASTE_CLASS_EXPORT(RodPhotoreceptor)
CHASTE_CLASS_EXPORT(AmacrineCell)
CHASTE_CLASS_EXPORT(BipolarCell)
CHASTE_CLASS_EXPORT(MullerGlia)
```

15.1.24 /src/GomesRetinalNeuralFates.hpp

```
#ifndef GOMESRETINALNEURALFATES_HPP_
#define GOMESRETINALNEURALFATES_HPP_

#include <boost/shared_ptr.hpp>
#include "AbstractCellProperty.hpp"
#include "ChasteSerialization.hpp"
#include <boost/serialization/base_object.hpp>

class RodPhotoreceptor : public AbstractCellProperty
{
private:

    /**
     * Colour for use by visualizer.
     */
    unsigned mColour;
```

```

    /** Needed for serialization. */
    friend class boost::serialization::access;
    /**
     * Archive the member variables.
     *
     * @param archive the archive
     * @param version the current version of this class
     */
    template<class Archive>
    void serialize(Archive & archive, const unsigned int version)
    {
        archive & boost::serialization::base_object<AbstractCellProperty>(*this);
        archive & mColour;
    }

public:

    /**
     * Constructor.
     *
     * @param colour  what colour cells with this property should be in the visualizer (defaults to 6)
     */
    RodPhotoreceptor(unsigned colour=3);

    /**
     * Destructor.
     */
    virtual ~RodPhotoreceptor();

    /**
     * @return #mColour.
     */
    unsigned GetColour() const;
};

class AmacrineCell : public AbstractCellProperty
{
private:

    /**
```

```
     * Colour for use by visualizer.
     */
    unsigned mColour;

    /** Needed for serialization. */
    friend class boost::serialization::access;
    /**
     * Archive the member variables.
     *
     * @param archive the archive
     * @param version the current version of this class
     */
    template<class Archive>
    void serialize(Archive & archive, const unsigned int version)
    {
        archive & boost::serialization::base_object<AbstractCellProperty>(*this);
        archive & mColour;
    }

public:

    /**
     * Constructor.
     *
     * @param colour  what colour cells with this property should be in the visualizer (defaults to 6)
     */
    AmacrineCell(unsigned colour=4);

    /**
     * Destructor.
     */
    virtual ~AmacrineCell();

    /**
     * @return #mColour.
     */
    unsigned GetColour() const;
};

class BipolarCell : public AbstractCellProperty
{
```

```
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 &
             ↪  boost::serialization::base_object<AbstractCellProperty>(*this);
118         archive & mColour;
119     }
120
121 public:
122
123     /**
124      * Constructor.
125      *
126      * @param colour  what colour cells with this property should be in
   ↪  the visualizer (defaults to 6)
127      */
128     BipolarCell(unsigned colour=5);
129
130     /**
131      * Destructor.
132      */
133     virtual ~BipolarCell();
134
135     /**
136      * @return #mColour.
137      */
138     unsigned GetColour() const;
139 };
```

```
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 &
            ↪  boost::serialization::base_object<AbstractCellProperty>(*this);
162         archive & mColour;
163     }
164
165 public:
166
167     /**
168      * Constructor.
169      *
170      * @param colour  what colour cells with this property should be in
    ↪  the visualizer (defaults to 6)
171      */
172     MullerGlia(unsigned colour=6);
173
174     /**
175      * Destructor.
176      */
177     virtual ~MullerGlia();
178
179     /**
180      * @return #mColour.
```

```
     */
    unsigned GetColour() const;
};
#include "SerializationExportWrapper.hpp"
// Declare identifier for the serializer
CHASTE_CLASS_EXPORT(RodPhotoreceptor)
CHASTE_CLASS_EXPORT(AmacrineCell)
CHASTE_CLASS_EXPORT(BipolarCell)
CHASTE_CLASS_EXPORT(MullerGlia)

#endif /* GOMESRETINALNEURALFATES_HPP_ */
```

15.1.25 /src/HeAth5Mo.cpp

```
#include "../../../../projects/ISP/src/HeAth5Mo.hpp"

Ath5Mo::Ath5Mo(unsigned colour)
    : AbstractCellProperty(),
      mColour(colour)
{
}

Ath5Mo::~Ath5Mo()
{
}

unsigned Ath5Mo::GetColour() const
{
    return mColour;
}

#include "SerializationExportWrapperForCpp.hpp"
// Declare identifier for the serializer
CHASTE_CLASS_EXPORT(Ath5Mo)
```

15.1.26 /src/HeAth5Mo.hpp

```
#ifndef HEATH5MO_HPP_
#define HEATH5MO_HPP_

#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      * 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 &
           ↪  boost::serialization::base_object<AbstractCellProperty>(*this);
30         archive & mColour;
31     }
32
33 public:
34
35     /**
36      * Constructor.
37      *
38      * @param colour  what colour cells with this property should be in
   ↪  the visualizer (defaults to 6)
39      */
40     Ath5Mo(unsigned colour=3);
41
42     /**
43      * Destructor.
44      */
45     virtual ~Ath5Mo();
```

```

    /**
     * @return #mColour.
     */
    unsigned GetColour() const;
};


#include "SerializationExportWrapper.hpp"
// Declare identifier for the serializer
CHASTE_CLASS_EXPORT(Ath5Mo)
#endif /* HEATH5MO_HPP */
```

15.1.27 /src/HeCellCycleModel.cpp

```
#include "HeCellCycleModel.hpp"

HeCellCycleModel::HeCellCycleModel() :
        AbstractSimpleCellCycleModel(), mKillSpecified(false), mDeterministic(false), mOutput(false), mEventStartTime(
                24.0), mSequenceSampler(false), mSeqSamplerLabelSister(false), mDebug(false), mTimeID(), mVarIDs(), mDebugWriter(), mTiLOffset(
                0.0), mGammaShift(4.0), mGammaShape(2.0), mGammaScale(1.0), mSisterShiftWidth(1), mMitoticModePhase2(
                8.0), mMitoticModePhase3(15.0), mPhaseShiftWidth(2.0), mPhase1PP(1.0), mPhase1PD(0.0), mPhase2PP(0.2), mPhase2PD(
                0.4), mPhase3PP(0.2), mPhase3PD(0.0), mMitoticMode(0), mSeed(0), mTimeDependentCycleDuration(false), mPeakRateTime(), mIncreasingRateSlope(), mDecreasingRateSlope(), mBaseGammaScale()
{
    mReadyToDivide = true; //He model begins with a first division
}

HeCellCycleModel::HeCellCycleModel(const HeCellCycleModel& rModel) :
        AbstractSimpleCellCycleModel(rModel), mKillSpecified(rModel.mKillSpecified), mDeterministic(
```

```
                rModel.mDeterministic), mOutput(rModel.mOutput), mEventStartTime(rModel.mEventStartTime), mSequenceSampler(
                rModel.mSequenceSampler), mSeqSamplerLabelSister(rModel.mSeqSamplerLabelSister), mDebug(rModel.mDebug), mTimeID(
                rModel.mTimeID), mVarIDs(rModel.mVarIDs), mDebugWriter(rModel.mDebugWriter), mTiLOffset(
                rModel.mTiLOffset), mGammaShift(rModel.mGammaShift), mGammaShape(rModel.mGammaShape), mGammaScale(
                rModel.mGammaScale), mSisterShiftWidth(rModel.mSisterShiftWidth), mMitoticModePhase2(
                rModel.mMitoticModePhase2), mMitoticModePhase3(rModel.mMitoticModePhase3), mPhaseShiftWidth(
                rModel.mPhaseShiftWidth), mPhase1PP(rModel.mPhase1PP), mPhase1PD(rModel.mPhase1PD), mPhase2PP(
                rModel.mPhase2PP), mPhase2PD(rModel.mPhase2PD), mPhase3PP(rModel.mPhase3PP), mPhase3PD(
                rModel.mPhase3PD), mMitoticMode(rModel.mMitoticMode), mSeed(rModel.mSeed), mTimeDependentCycleDuration(
                rModel.mTimeDependentCycleDuration), mPeakRateTime(rModel.mPeakRateTime), mIncreasingRateSlope(
                rModel.mIncreasingRateSlope), mDecreasingRateSlope(rModel.mDecreasingRateSlope), mBaseGammaScale(
                rModel.mBaseGammaScale)
{
}

AbstractCellCycleModel* HeCellCycleModel::CreateCellCycleModel()
{
    return new HeCellCycleModel(*this);
}

void HeCellCycleModel::SetCellCycleDuration()
{
    RandomNumberGenerator* p_random_number_generator = RandomNumberGenerator::Instance();

    /*************************************
```

```
     * CELL CYCLE DURATION RANDOM VARIABLE
     ************************************/

    if (!mTimeDependentCycleDuration) //Normal operation, cell cycle length stays constant
    {
        //He cell cycle length determined by shifted gamma distribution reflecting 4 hr refractory period followed by gamma pdf
        mCellCycleDuration = mGammaShift + p_random_number_generator->GammaRandomDeviate(mGammaShape, mGammaScale);
    }

    /****
     * Variable cycle length
     * Give -ve mIncreasingRateSlope and +ve mDecreasingRateSlope,
     * cell cycle length linearly declines (increasing rate), then increases, switching at mPeakRateTime
     ****/
    else
    {
        double currTime = SimulationTime::Instance()->GetTime();
        if (currTime <= mPeakRateTime)
        {
            mGammaScale = std::max((mBaseGammaScale - currTime * mIncreasingRateSlope), .0000000000001);
        }
        if (currTime > mPeakRateTime)
        {
            mGammaScale = std::max(
                    ((mBaseGammaScale - mPeakRateTime * mIncreasingRateSlope)
                            + (mBaseGammaScale + (currTime - mPeakRateTime) * mDecreasingRateSlope)),
                    .0000000000001);
        }
        mCellCycleDuration = mGammaShift + p_random_number_generator->GammaRandomDeviate(mGammaShape, mGammaScale);
    }

}
```

```
73 void HeCellCycleModel::ResetForDivision()
74 {
75     /**************************************************
76      * TIME IN LINEAGE DEPENDENT MITOTIC MODE PHASE RULES
77      * **************************************************/
78     RandomNumberGenerator* p_random_number_generator =
        ↪  RandomNumberGenerator::Instance();
79
80     double currentTiL = SimulationTime::Instance()->GetTime() +
        ↪  mTiLOffset;
81
82     /*Rule logic defaults to phase 1 behaviour, checks for currentTiL >
        ↪  phaseBoundaries and changes
83      currentPhase and subsequently mMitoticMode as appropriate*/
84     unsigned currentPhase = 1;
85     mMitoticMode = 0;
86
87     //Check time in lineage and determine current mitotic mode phase
88     /**************
89      * Phase boundary & deterministic mitotic mode rules
90      **************/
91     if (currentTiL > mMitoticModePhase2 && currentTiL <
        ↪  mMitoticModePhase3)
92     {
93         //if current TiL is > phase 2 boundary time, set the currentPhase
            ↪  appropriately
94         currentPhase = 2;
95
96         //if deterministic mode is enabled, PD divisions are guaranteed
            ↪  unless this is an Ath5 morphant
97         if (mDeterministic)
98         {
99             mMitoticMode = 1; //0=PP;1=PD;2=DD
100            if (mpCell->HasCellProperty<Ath5Mo>()) //Ath5 morphants
                ↪  undergo PP rather than PD divisions in 80% of cases
101            {
102                double ath5RV = p_random_number_generator->ranf();
103                if (ath5RV <= .8)
104                {
105                    mMitoticMode = 0;
106                }
107            }
108        }
```

```
109     }
110
111     if (currentTiL > mMitoticModePhase3)
112     {
113         //if current TiL is > phase 3 boundary time, set the currentPhase
            ↪  appropriately
114         currentPhase = 3;
115         if (mDeterministic)
116         {
117             //if deterministic mode is enabled, DD divisions are
                ↪  guaranteed
118             mMitoticMode = 2;
119         }
120     }
121
122     /******************************
123      * MITOTIC MODE RANDOM VARIABLE
124      ******************************/
125     //initialise mitoticmode random variable, set mitotic mode
        ↪  appropriately after comparing to mode probability matrix
126     double mitoticModeRV = p_random_number_generator→ranf(); //0-1
        ↪  evenly distributed RV
127
128     if (!mDeterministic)
129     {
130         //construct 3×2 matrix of mode probabilities arranged by phase
131         double modeProbabilityMatrix[3][2] = { { mPhase1PP, mPhase1PD },
            ↪  { mPhase2PP, mPhase2PD }, { mPhase3PP,
132
133
134         //if the RV is > currentPhasePP && ≤ currentPhasePD, change
            ↪  mMitoticMode from PP to PD
135         if (mitoticModeRV > modeProbabilityMatrix[currentPhase - 1][0]
136                 && mitoticModeRV
137                         ≤ modeProbabilityMatrix[currentPhase - 1][0] +
                            ↪  modeProbabilityMatrix[currentPhase - 1][1])
138         {
139             mMitoticMode = 1;
140             if (mpCell→HasCellProperty<Ath5Mo>()) //Ath5 morphants
                ↪  undergo PP rather than PD divisions in 80% of cases
```

```
            {
                double ath5RV = p_random_number_generator->ranf();
                if (ath5RV <= .8)
                {
                    mMitoticMode = 0;
                }
            }
        }
        //if the RV is > currentPhasePP + currentPhasePD, change mMitoticMode from PP to DD
        if (mitoticModeRV > modeProbabilityMatrix[currentPhase - 1][0] + modeProbabilityMatrix[currentPhase - 1][1])
        {
            mMitoticMode = 2;
        }
    }

    /****************
     * Write mitotic event to relevant files
     * *************/
    if (mDebug)
    {
        WriteDebugData(currentTiL, currentPhase, mitoticModeRV);
    }

    if (mOutput)
    {
        WriteModeEventOutput();
    }

    //set new cell cycle length (will be overwritten with DBL_MAX for DD divisions)
    AbstractSimpleCellCycleModel::ResetForDivision();

    /****************
     * Symmetric postmitotic specification rule
     * -(asymmetric postmitotic rule specified in InitialiseDaughterCell();)
     * *************/
    if (mMitoticMode == 2)
    {
        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 &
         ↪  write mitotic mode to log
193     //50% chance of each daughter cell from a mitosis inheriting the
         ↪  label
194     if (mSequenceSampler)
195     {
196         if (mpCell→HasCellProperty<CellLabel>())
197         {
198             (*LogFile::Instance()) << mMitoticMode;
199             double labelRV = p_random_number_generator→ranf();
200             if (labelRV ≤ .5)
201             {
202                 mSeqSamplerLabelSister = true;
203                 mpCell→RemoveCellProperty<CellLabel>();
204             }
205             else
206             {
207                 mSeqSamplerLabelSister = false;
208             }
209         }
210         else
211         {
212             //prevents lost-label cells from labelling their progeny
213             mSeqSamplerLabelSister = false;
214         }
215     }
216 }
217
218 void HeCellCycleModel::Initialise()
```

```
{
    boost::shared_ptr<AbstractCellProperty> p_Transit =
            ↪ mpCell->rGetCellPropertyCollection().GetCellPropertyRegistry()->Get<T
    mpCell->SetCellProliferativeType(p_Transit);

    if (mTiLOffset == 0) //the "regular" case, set cycle duration
     ↪ normally
    {
        SetCellCycleDuration();
    }

    if (mTiLOffset < 0) //these cells are offspring of Wan stem cells
    {
        mReadyToDivide = false;
        SetCellCycleDuration();
    }

    if (mTiLOffset > 0.0) //if the TiL is > 0, the first division has
     ↪ already occurred
    {
        mReadyToDivide = false;

        RandomNumberGenerator* p_random_number_generator =
         ↪ RandomNumberGenerator::Instance();

        /**
         * This calculation "runs time forward" by subtracting
↪ appropriately generated cell lengths from TiLOffset
         * Ultimately c is subtracted from a final cell length
↪ calculation to give the appropriate reduced cycle length
         **/

        double c = mTiLOffset;
        while (c > 0)
        {
            c = c - (mGammaShift +
             ↪ p_random_number_generator->GammaRandomDeviate(mGammaShape,
             ↪ mGammaScale));
        }

```

```
        mCellCycleDuration = (mGammaShift + p_random_number_generator->GammaRandomDeviate(mGammaShape, mGammaScale))
                + c;
    }

}

void HeCellCycleModel::InitialiseDaughterCell()
{
    RandomNumberGenerator* p_random_number_generator = RandomNumberGenerator::Instance();

    /************
     * PD-type division & shifted sister cycle length & boundary adjustments
     **********/

    if (mMitoticMode == 1) //RPC becomes specified retinal neuron in asymmetric PD mitosis
    {
        boost::shared_ptr<AbstractCellProperty> p_PostMitoticType = mpCell->rGetCellPropertyCollection().GetCellPropertyRegistry()->G
        mpCell->SetCellProliferativeType(p_PostMitoticType);
        mCellCycleDuration = DBL_MAX;

        if(mKillSpecified)
        {
            mpCell->Kill();
        }
    }

    //daughter cell's mCellCycleDuration is copied from parent; modified by a normally distributed shift if it remains proliferative
    if (mMitoticMode == 0)
    {
        double sisterShift = p_random_number_generator->NormalRandomDeviate(0, mSisterShiftWidth); //random variable mean 0 SD 1 by default
        mCellCycleDuration = std::max(mGammaShift, mCellCycleDuration + sisterShift); // sister shift respects 4 hour refractory period
```

```
    }

    //deterministic model phase boundary division shift for daughter cells
    if (mDeterministic)
    {
        //shift phase boundaries to reflect error in "timer" after division
        double phaseShift = p_random_number_generator->NormalRandomDeviate(0, mPhaseShiftWidth);
        mMitoticModePhase2 = mMitoticModePhase2 + phaseShift;
        mMitoticModePhase3 = mMitoticModePhase3 + phaseShift;
    }

    /******************
     * SEQUENCE SAMPLER
     ******************/
    if (mSequenceSampler)
    {
        if (mSeqSamplerLabelSister)
        {
            boost::shared_ptr<AbstractCellProperty> p_label_type = mpCell->rGetCellPropertyCollection().GetCellPropertyRegistry(
            mpCell->AddCellProperty(p_label_type);
            mSeqSamplerLabelSister = false;
        }
        else
        {
            mpCell->RemoveCellProperty<CellLabel>();
        }
    }

    if (mMitoticMode == 2 && mKillSpecified) mpCell->Kill();
}

void HeCellCycleModel::SetModelParameters(double tiLOffset, double mitoticModePhase2, double mitoticModePhase3,
                                          double phase1PP, double phase1PD, double phase2PP, double phase2PD,
```

```
                                                double phase3PP, double phase3PD, double gammaShift, double gammaShape,
                                                double gammaScale, double sisterShift)
{
    mTiLOffset = tiLOffset;
    mMitoticModePhase2 = mitoticModePhase2;
    mMitoticModePhase3 = mitoticModePhase3;
    mPhase1PP = phase1PP;
    mPhase1PD = phase1PD;
    mPhase2PP = phase2PP;
    mPhase2PD = phase2PD;
    mPhase3PP = phase3PP;
    mPhase3PD = phase3PD;
    mGammaShift = gammaShift;
    mGammaShape = gammaShape;
    mGammaScale = gammaScale;
    mSisterShiftWidth = sisterShift;
}

void HeCellCycleModel::SetDeterministicMode(double tiLOffset, double mitoticModePhase2, double mitoticModePhase3,
                                                double phaseShiftWidth, double gammaShift, double gammaShape,
                                                double gammaScale, double sisterShift)
{
    mDeterministic = true;
    mTiLOffset = tiLOffset;
    mMitoticModePhase2 = mitoticModePhase2;
    mMitoticModePhase3 = mitoticModePhase3;
    mPhaseShiftWidth = phaseShiftWidth;
    mGammaShift = gammaShift;
    mGammaShape = gammaShape;
    mGammaScale = gammaScale;
    mSisterShiftWidth = sisterShift;
}

void HeCellCycleModel::SetTimeDependentCycleDuration(double peakRateTime, double increasingSlope,
```

```
                                                                double
                                                                ↪ decreasingSlope)
{
    mTimeDependentCycleDuration = true;
    mPeakRateTime = peakRateTime;
    mIncreasingRateSlope = increasingSlope;
    mDecreasingRateSlope = decreasingSlope;
    mBaseGammaScale = mGammaScale;
}

void HeCellCycleModel::EnableKillSpecified()
{
    mKillSpecified = true;
}

void HeCellCycleModel::EnableModeEventOutput(double eventStart, unsigned
↪ seed)
{
    mOutput = true;
    mEventStartTime = eventStart;
    mSeed = seed;
}

void HeCellCycleModel::WriteModeEventOutput()
{
    double currentTime = SimulationTime::Instance()->GetTime() +
    ↪ mEventStartTime;
    CellPtr currentCell = GetCell();
    double currentCellID = (double) currentCell->GetCellId();
    (*LogFile::Instance()) << currentTime << "\t" << mSeed << "\t" <<
    ↪ currentCellID << "\t" << mMitoticMode << "\n";
}

void HeCellCycleModel::EnableSequenceSampler()
{
    mSequenceSampler = true;
    boost::shared_ptr<AbstractCellProperty> p_label_type =

            ↪ mpCell->rGetCellPropertyCollection().GetCellPropertyRegistry()->Get<C
    mpCell->AddCellProperty(p_label_type);
}

```

```
void HeCellCycleModel::PassDebugWriter(boost::shared_ptr<ColumnDataWriter> debugWriter, int timeID,
                                       std::vector<int> varIDs)
{
    mDebug = true;
    mDebugWriter = debugWriter;
    mTimeID = timeID;
    mVarIDs = varIDs;
}

void HeCellCycleModel::EnableModelDebugOutput(boost::shared_ptr<ColumnDataWriter> debugWriter)
{
    mDebug = true;
    mDebugWriter = debugWriter;

    mTimeID = mDebugWriter->DefineUnlimitedDimension("Time", "h");

    mVarIDs.push_back(mDebugWriter->DefineVariable("CellID", "No"));
    mVarIDs.push_back(mDebugWriter->DefineVariable("TiL", "h"));
    mVarIDs.push_back(mDebugWriter->DefineVariable("CycleDuration", "h"));
    mVarIDs.push_back(mDebugWriter->DefineVariable("Phase2Boundary", "h"));
    mVarIDs.push_back(mDebugWriter->DefineVariable("Phase3Boundary", "h"));
    mVarIDs.push_back(mDebugWriter->DefineVariable("Phase", "No"));
    mVarIDs.push_back(mDebugWriter->DefineVariable("MitoticModeRV", "Percentile"));
    mVarIDs.push_back(mDebugWriter->DefineVariable("MitoticMode", "Mode"));
    mVarIDs.push_back(mDebugWriter->DefineVariable("Label", "binary"));

    mDebugWriter->EndDefineMode();
}

void HeCellCycleModel::WriteDebugData(double currentTiL, unsigned phase, double mitoticModeRV)
{
    double currentTime = SimulationTime::Instance()->GetTime();
    double currentCellID = mpCell->GetCellId();
```

```
    unsigned label = 0;
    if (mpCell->HasCellProperty<CellLabel>()) label = 1;

    mDebugWriter->PutVariable(mTimeID, currentTime);
    mDebugWriter->PutVariable(mVarIDs[0], currentCellID);
    mDebugWriter->PutVariable(mVarIDs[1], currentTiL);
    mDebugWriter->PutVariable(mVarIDs[2], mCellCycleDuration);
    mDebugWriter->PutVariable(mVarIDs[3], mMitoticModePhase2);
    mDebugWriter->PutVariable(mVarIDs[4], mMitoticModePhase3);
    mDebugWriter->PutVariable(mVarIDs[5], phase);
    if (!mDeterministic)
    {
        mDebugWriter->PutVariable(mVarIDs[6], mitoticModeRV);
    }
    mDebugWriter->PutVariable(mVarIDs[7], mMitoticMode);
    if (mSequenceSampler)
    {
        mDebugWriter->PutVariable(mVarIDs[8], label);
    }
    mDebugWriter->AdvanceAlongUnlimitedDimension();
}

/******************
 * UNUSED FUNCTIONS (required for class nonvirtuality, do not remove)
 ******************/

double HeCellCycleModel::GetAverageTransitCellCycleTime()
{
    return (0.0);
}

double HeCellCycleModel::GetAverageStemCellCycleTime()
{
    return (0.0);
}

void HeCellCycleModel::OutputCellCycleModelParameters(out_stream&
    rParamsFile)
{
    *rParamsFile << "\t\t\t<CellCycleDuration>" << mCellCycleDuration <<
        "</CellCycleDuration>\n";

    // Call method on direct parent class
```

```
    ↪ AbstractSimpleCellCycleModel::OutputCellCycleModelParameters(rParamsFile);
}

// Serialization for Boost ≥ 1.36
#include "SerializationExportWrapperForCpp.hpp"
CHASTE_CLASS_EXPORT(HeCellCycleModel)
```

15.1.28 /src/HeCellCycleModel.hpp

```
#ifndef HECELLCYCLEMODEL_HPP_
#define HECELLCYCLEMODEL_HPP_

#include "AbstractSimpleCellCycleModel.hpp"
#include "RandomNumberGenerator.hpp"
#include "Cell.hpp"
#include "TransitCellProliferativeType.hpp"
#include "DifferentiatedCellProliferativeType.hpp"
#include "SmartPointers.hpp"
#include "ColumnDataWriter.hpp"
#include "LogFile.hpp"
#include "CellLabel.hpp"
#include "HeAth5Mo.hpp"

/***********************************
 * HE CELL CYCLE MODEL
 * As described in He et al. 2012 [He2012] doi:
 ↪ 10.1016/j.neuron.2012.06.033
 *
 * USE: By default, HeCellCycleModels are constructed with the parameter
 ↪ fit reported in [He2012].
 * In normal use, the model steps through three phases of mitotic mode
 ↪ probability parameterisation.
 * PP = symmetric proliferative mitotic mode, both progeny remain mitotic
 * PD = asymmetric proliferative mitotic mode, one progeny exits the cell
 ↪ cycle and differentiates
 * DD = symmetric differentiative mitotic mode, both progeny exit the
 ↪ cell cycle and differentiate
 *
 * Change default model parameters with SetModelParameters(<params>);
 * Enable deterministic model alternative with
 ↪ EnableDeterministicMode(<params>);
```

```
 *
 * 2 per-model-event output modes:
 * EnableModeEventOutput() enables mitotic mode event logging-all cells will write to the singleton log file
 * EnableModelDebugOutput() enables more detailed debug output, each seed will have its own file written to
 * by a ColumnDataWriter passed to it from the test
 * (eg. by the SetupDebugOutput helper function in the project simulator)
 *
 * 1 mitotic-event-sequence sampler (only samples one "path" through the lineage):
 * EnableSequenceSampler() - one "sequence" of progenitors writes mitotic event type to a string in the singleton log file
 *
 ************************************/

class HeCellCycleModel : public AbstractSimpleCellCycleModel
{
    friend class TestSimpleCellCycleModels;

private:

    /** Needed for serialization. */
    friend class boost::serialization::access;
    /**
     * Archive the cell-cycle model and random number generator, never used directly - boost uses this.
     *
     * @param archive the archive
     * @param version the current version of this class
     */
    template<class Archive>
    void serialize(Archive & archive, const unsigned int version)
    {
        archive & boost::serialization::base_object<AbstractSimpleCellCycleModel>(*this);

        SerializableSingleton<RandomNumberGenerator>* p_wrapper = RandomNumberGenerator::Instance()->GetSerializationWrapper();
        archive & p_wrapper;
        archive & mCellCycleDuration;
    }
```

```
63
64      //Private write functions for models
65      void WriteModeEventOutput();
66      void WriteDebugData(double currTiL, unsigned phase, double
        ↪  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
    ↪  model
108      * is by calling that method, to ensure that a model of the correct
    ↪  subclass is created.
109      * This copy-constructor helps subclasses to ensure that all member
    ↪  variables are correctly copied when this happens.
110      *
111      * This method is called by child classes to set member variables for
    ↪  a daughter cell upon cell division.
112      * Note that the parent cell cycle model will have had
    ↪  ResetForDivision() called just before CreateCellCycleModel() is
    ↪  called,
113      * so performing an exact copy of the parent is suitable behaviour.
    ↪  Any daughter-cell-specific initialisation
114      * can be done in InitialiseDaughterCell().
115      *
116      * @param rModel the cell cycle model to copy.
117      */
118     HeCellCycleModel(const HeCellCycleModel& rModel);
119
120 public:
121
122     /**
123      * Constructor - just a default, mBirthTime is set in the
    ↪  AbstractCellCycleModel class.
124      */
125     HeCellCycleModel();
126
127     /**
128      * SetCellCycleDuration() method to set length of cell cycle
129      */
130     void SetCellCycleDuration();
131
132     /**
133      * Overridden builder method to create new copies of
134      * this cell-cycle model.
135      *
136      * @return new cell-cycle model
137      */
138     AbstractCellCycleModel* CreateCellCycleModel();
139
```

```
    /**
     * Overridden ResetForDivision() method.
     * Contains general mitotic mode logic
     **/
    void ResetForDivision();

    /**
     * Overridden Initialise() method
     * Used to give an appropriate mCellCycleDuration to cells w/ TiL offsets
     * sets mReadytoDivide to false as appropriate
     * Initialises as transit proliferative type
     **/
    void Initialise();

    /**
     * Overridden InitialiseDaughterCell() method.
     * Used to apply sister-cell time shifting (cell cycle duration, deterministic phase boundaries)
     * Used to implement asymmetric mitotic mode
     * */
    void InitialiseDaughterCell();

    /*Model setup functions for standard He (SetModelParameters) and deterministic alternative (SetDeterministicMode) models
     * Default parameters are from refits of He et al + deterministic alternatives
     * He 2012 params: mitoticModePhase2 = 8, mitoticModePhase3 = 15, p1PP = 1, p1PD = 0, p2PP = .2, p2PD = .4, p3PP = .2, p3PD = 0
     * gammaShift = 4, gammaShape = 2, gammaScale = 1, sisterShift = 1
     */
    void SetModelParameters(double tiLOffset = 0, double mitoticModePhase2 = 8, double mitoticModePhase3 = 15,
                            double phase1PP = 1, double phase1PD = 0, double phase2PP = .2, double phase2PD = .4,
                            double phase3PP = .2, double phase3PD = 0, double gammaShift = 4, double gammaShape = 2,
                            double gammaScale = 1, double sisterShift = 1);
    void SetDeterministicMode(double tiLOffset = 0, double mitoticModePhase2 = 8, double mitoticModePhase3 = 15,
```

```
                                double phaseShiftWidth = 1, double gammaShift = 4, double gammaShape = 2,
                                double gammaScale = 1, double sisterShift = 1);
    void SetTimeDependentCycleDuration(double peakRateTime, double increasingSlope, double decreasingSlope);

    //Function to set mKillSpecified = true; marks specified neurons for death and removal from population
    //Intended to help w/ resource consumption for WanSimulator
    void EnableKillSpecified();

    //Functions to enable per-cell mitotic mode logging for mode rate & sequence sampling fixtures
    //Uses singleton logfile
    void EnableModeEventOutput(double eventStart, unsigned seed);
    void EnableSequenceSampler();

    //More detailed debug output. Needs a ColumnDataWriter passed to it
    //Only declare ColumnDataWriter directory, filename, etc; do not set up otherwise
    //Use PassDebugWriter if the writer is already enabled elsewhere (ie. in a Wan stem cell cycle model)
    void EnableModelDebugOutput(boost::shared_ptr<ColumnDataWriter> debugWriter);
    void PassDebugWriter(boost::shared_ptr<ColumnDataWriter> debugWriter, int timeID, std::vector<int> varIDs);

    //Not used, but must be overwritten lest HeCellCycleModels be abstract
    double GetAverageTransitCellCycleTime();
    double GetAverageStemCellCycleTime();

    /**
     * Overridden OutputCellCycleModelParameters() method.
     *
     * @param rParamsFile the file stream to which the parameters are output
     */
    virtual void OutputCellCycleModelParameters(out_stream& rParamsFile);
};

#include "SerializationExportWrapper.hpp"
```

```
// Declare identifier for the serializer
CHASTE_CLASS_EXPORT(HeCellCycleModel)

#endif /*HECELLCYCLEMODEL_HPP_*/
```

15.1.29 /src/OffLatticeSimulationPropertyStop.cpp

```
#include "OffLatticeSimulationPropertyStop.hpp"

#include <boost/make_shared.hpp>

#include "CellBasedEventHandler.hpp"
#include "ForwardEulerNumericalMethod.hpp"
#include "StepSizeException.hpp"

template<unsigned ELEMENT_DIM, unsigned SPACE_DIM>
OffLatticeSimulationPropertyStop<ELEMENT_DIM,SPACE_DIM>::OffLatticeSimulationPrope
    rCellPopulation,
                                                bool
                                                    deleteCellPopulationInDestructor,
                                                bool initialiseCells
                                                )
    : AbstractCellBasedSimulation<ELEMENT_DIM,SPACE_DIM>(rCellPopulation,
        deleteCellPopulationInDestructor, initialiseCells),
    p_property()
{
    if
        (!dynamic_cast<AbstractOffLatticeCellPopulation<ELEMENT_DIM,SPACE_DIM>*>(&rCe
    {
        EXCEPTION("OffLatticeSimulations require a subclass of
            AbstractOffLatticeCellPopulation.");
    }
}

template<unsigned ELEMENT_DIM, unsigned SPACE_DIM>
bool
    OffLatticeSimulationPropertyStop<ELEMENT_DIM,SPACE_DIM>::StoppingEventHasOccurred
{
    if(p_property->GetCellCount()<1){
                return true;
        }
        else{
```

```
                return false;
        }
}

//Public access to Stopping Event bool
template<unsigned ELEMENT_DIM, unsigned SPACE_DIM>
bool
↪ OffLatticeSimulationPropertyStop<ELEMENT_DIM,SPACE_DIM>::HasStoppingEventOccurred
{
    return StoppingEventHasOccurred();
}

template<unsigned ELEMENT_DIM, unsigned SPACE_DIM>
void
↪ OffLatticeSimulationPropertyStop<ELEMENT_DIM,SPACE_DIM>::SetStopProperty(boost::s
↪ stopPropertySetting)
{
        p_property = stopPropertySetting;
}

template<unsigned ELEMENT_DIM, unsigned SPACE_DIM>
void
↪ OffLatticeSimulationPropertyStop<ELEMENT_DIM,SPACE_DIM>::AddForce(boost::shared_p
↪ > pForce)
{
    mForceCollection.push_back(pForce);
}

template<unsigned ELEMENT_DIM, unsigned SPACE_DIM>
void
↪ OffLatticeSimulationPropertyStop<ELEMENT_DIM,SPACE_DIM>::RemoveAllForces()
{
    mForceCollection.clear();
}

template<unsigned ELEMENT_DIM, unsigned SPACE_DIM>
void
↪ OffLatticeSimulationPropertyStop<ELEMENT_DIM,SPACE_DIM>::AddCellPopulationBoundar
↪ > pBoundaryCondition)
{
    mBoundaryConditions.push_back(pBoundaryCondition);
}

```

```
template<unsigned ELEMENT_DIM, unsigned SPACE_DIM>
void
↪ OffLatticeSimulationPropertyStop<ELEMENT_DIM,SPACE_DIM>::RemoveAllCellPopulationB
{
    mBoundaryConditions.clear();
}

template<unsigned ELEMENT_DIM, unsigned SPACE_DIM>
void
↪ OffLatticeSimulationPropertyStop<ELEMENT_DIM,SPACE_DIM>::SetNumericalMethod(boost
↪ SPACE_DIM> > pNumericalMethod)
{
    mpNumericalMethod = pNumericalMethod;
}

template<unsigned ELEMENT_DIM, unsigned SPACE_DIM>
const boost::shared_ptr<AbstractNumericalMethod<ELEMENT_DIM, SPACE_DIM> >
↪ OffLatticeSimulationPropertyStop<ELEMENT_DIM,SPACE_DIM>::GetNumericalMethod()
↪ const
{
    return mpNumericalMethod;
}

template<unsigned ELEMENT_DIM, unsigned SPACE_DIM>
const std::vector<boost::shared_ptr<AbstractForce<ELEMENT_DIM, SPACE_DIM>
↪ > >&
↪ OffLatticeSimulationPropertyStop<ELEMENT_DIM,SPACE_DIM>::rGetForceCollection()
↪ const
{
    return mForceCollection;
}

template<unsigned ELEMENT_DIM, unsigned SPACE_DIM>
void
↪ OffLatticeSimulationPropertyStop<ELEMENT_DIM,SPACE_DIM>::UpdateCellLocationsAndTo
{
    CellBasedEventHandler::BeginEvent(CellBasedEventHandler::POSITION);

    double time_advanced_so_far = 0;
    double target_time_step  = this->mDt;
    double present_time_step = this->mDt;

    while (time_advanced_so_far < target_time_step)
```

```
    {
        // Store the initial node positions (these may be needed when applying boundary conditions)
        std::map<Node<SPACE_DIM>*, c_vector<double, SPACE_DIM> > old_node_locations;

        for (typename AbstractMesh<ELEMENT_DIM, SPACE_DIM>::NodeIterator node_iter = this->mrCellPopulation.rGetMesh().GetNodeIteratorBegin();
             node_iter != this->mrCellPopulation.rGetMesh().GetNodeIteratorEnd();
             ++node_iter)
        {
            old_node_locations[&(*node_iter)] = (node_iter)->rGetLocation();
        }

        // Try to update node positions according to the numerical method
        try
        {
            mpNumericalMethod->UpdateAllNodePositions(present_time_step);
            ApplyBoundaries(old_node_locations);

            // Successful time step! Update time_advanced_so_far
            time_advanced_so_far += present_time_step;

            // If using adaptive timestep, then increase the present_time_step (by 1% for now)
            if (mpNumericalMethod->HasAdaptiveTimestep())
            {
                ///\todo #2087 Make this a settable member variable
                double timestep_increase = 0.01;
                present_time_step = std::min((1+timestep_increase)*present_time_step, target_time_step - time_advanced_so_far);
            }

        }
        catch (StepSizeException& e)
        {
            // Detects if a node has travelled too far in a single time step
            if (mpNumericalMethod->HasAdaptiveTimestep())
```

```
            {
                // If adaptivity is switched on, revert node locations and choose a suitably smaller time step
                RevertToOldLocations(old_node_locations);
                present_time_step = std::min(e.GetSuggestedNewStep(), target_time_step - time_advanced_so_far);
            }
            else
            {
                // If adaptivity is switched off, terminate with an error
                EXCEPTION(e.what());
            }
        }
    }

    CellBasedEventHandler::EndEvent(CellBasedEventHandler::POSITION);
}

template<unsigned ELEMENT_DIM, unsigned SPACE_DIM>
void OffLatticeSimulationPropertyStop<ELEMENT_DIM,SPACE_DIM>::RevertToOldLocations(std::c_vector<double, SPACE_DIM> > oldNodeLoctions)
{
    for (typename AbstractMesh<ELEMENT_DIM, SPACE_DIM>::NodeIterator node_iter = this->mrCellPopulation.rGetMesh().GetNodeIteratorBegin();
        node_iter != this->mrCellPopulation.rGetMesh().GetNodeIteratorEnd();
        ++node_iter)
    {
        (node_iter)->rGetModifiableLocation() = oldNodeLoctions[&(*node_iter)];
    }
}

template<unsigned ELEMENT_DIM, unsigned SPACE_DIM>
void OffLatticeSimulationPropertyStop<ELEMENT_DIM,SPACE_DIM>::ApplyBoundaries(std::map SPACE_DIM> > oldNodeLoctions)
{
    // Apply any boundary conditions
```

```
    for (typename std::vector<boost::shared_ptr<AbstractCellPopulationBoundaryCondition<ELEMENT_
    > >::iterator bcs_iter = mBoundaryConditions.begin();
         bcs_iter != mBoundaryConditions.end();
         ++bcs_iter)
    {
        (*bcs_iter)->ImposeBoundaryCondition(oldNodeLoctions);
    }

    // Verify that each boundary condition is now satisfied
    for (typename std::vector<boost::shared_ptr<AbstractCellPopulationBoundaryCondition<ELEMENT_
    > >::iterator bcs_iter = mBoundaryConditions.begin();
         bcs_iter != mBoundaryConditions.end();
         ++bcs_iter)
    {
        if (!((*bcs_iter)->VerifyBoundaryCondition()))
        {
            EXCEPTION("The cell population boundary conditions are incompatible.");
        }
    }
}

template<unsigned ELEMENT_DIM, unsigned SPACE_DIM>
void OffLatticeSimulationPropertyStop<ELEMENT_DIM,SPACE_DIM>::WriteVisualizerSetupFile
{
    if (PetscTools::AmMaster())
    {
        for (unsigned i=0; i<this->mForceCollection.size(); i++)
        {
            this->mForceCollection[i]->WriteDataToVisualizerSetupFile(this->mpViz
        }


        this->mrCellPopulation.WriteDataToVisualizerSetupFile(this->mpVizSetupFil
    }
}

template<unsigned ELEMENT_DIM, unsigned SPACE_DIM>
```

```
197 void
    ↪ OffLatticeSimulationPropertyStop<ELEMENT_DIM,SPACE_DIM>::SetupSolve()
198 {
199     // Clear all forces
200     for (typename AbstractMesh<ELEMENT_DIM, SPACE_DIM>::NodeIterator
        ↪ node_iter =
        ↪ this->mrCellPopulation.rGetMesh().GetNodeIteratorBegin();
201         node_iter !=
            ↪ this->mrCellPopulation.rGetMesh().GetNodeIteratorEnd();
202         ++node_iter)
203     {
204         node_iter->ClearAppliedForce();
205     }
206
207     // Use a forward Euler method by default, unless a numerical method
        ↪ has been specified already
208     if (mpNumericalMethod == nullptr)
209     {
210         mpNumericalMethod =
            ↪ boost::make_shared<ForwardEulerNumericalMethod<ELEMENT_DIM,
            ↪ SPACE_DIM> >();
211     }
212
        ↪ mpNumericalMethod->SetCellPopulation(dynamic_cast<AbstractOffLatticeCellPopul
213     mpNumericalMethod->SetForceCollection(&mForceCollection);
214 }
215
216 template<unsigned ELEMENT_DIM, unsigned SPACE_DIM>
217 void
    ↪ OffLatticeSimulationPropertyStop<ELEMENT_DIM,SPACE_DIM>::OutputAdditionalSimulati
    ↪ rParamsFile)
218 {
219     // Loop over forces
220     *rParamsFile << "\n\t<Forces>\n";
221     for (typename
        ↪ std::vector<boost::shared_ptr<AbstractForce<ELEMENT_DIM,SPACE_DIM>
        ↪ > >::iterator iter = mForceCollection.begin();
222         iter != mForceCollection.end();
223         ++iter)
224     {
225         // Output force details
226         (*iter)->OutputForceInfo(rParamsFile);
227     }
```

```
    *rParamsFile << "\t</Forces>\n";

    // Loop over cell population boundary conditions
    *rParamsFile << "\n\t<CellPopulationBoundaryConditions>\n";
    for (typename std::vector<boost::shared_ptr<AbstractCellPopulationBoundaryCondition<ELEMENT_
        iter mBoundaryConditions.begin();
        iter ≠ mBoundaryConditions.end();
        ++iter)
    {
        // Output cell boundary condition details
        (*iter)→OutputCellPopulationBoundaryConditionInfo(rParamsFile);
    }
    *rParamsFile << "\t</CellPopulationBoundaryConditions>\n";

    // Output numerical method details
    *rParamsFile << "\n\t<NumericalMethod>\n";
    mpNumericalMethod→OutputNumericalMethodInfo(rParamsFile);
    *rParamsFile << "\t</NumericalMethod>\n";
}

template<unsigned ELEMENT_DIM, unsigned SPACE_DIM>
void OffLatticeSimulationPropertyStop<ELEMENT_DIM,SPACE_DIM>::OutputSimulationParamete
    rParamsFile)
{
    // No new parameters to output, so just call method on direct parent class

    AbstractCellBasedSimulation<ELEMENT_DIM,SPACE_DIM>::OutputSimulationParameter
}

// Explicit instantiation
template class OffLatticeSimulationPropertyStop<1,1>;
template class OffLatticeSimulationPropertyStop<1,2>;
template class OffLatticeSimulationPropertyStop<2,2>;
template class OffLatticeSimulationPropertyStop<1,3>;
template class OffLatticeSimulationPropertyStop<2,3>;
template class OffLatticeSimulationPropertyStop<3,3>;

// Serialization for Boost ≥ 1.36
#include "SerializationExportWrapperForCpp.hpp"
EXPORT_TEMPLATE_CLASS_ALL_DIMS(OffLatticeSimulationPropertyStop)
```

15.1.30 /src/OffLatticeSimulationPropertyStop.hpp

```
#ifndef OFFLATTICESIMULATIONPROPERTYSTOP_HPP_
#define OFFLATTICESIMULATIONPROPERTYSTOP_HPP_

#include "AbstractCellBasedSimulation.hpp"
#include "AbstractForce.hpp"
#include "AbstractCellPopulationBoundaryCondition.hpp"
#include "AbstractNumericalMethod.hpp"

#include "ChasteSerialization.hpp"
#include <boost/serialization/base_object.hpp>
#include <boost/serialization/set.hpp>
#include <boost/serialization/vector.hpp>

/**
 * Run an off-lattice 2D or 3D cell-based simulation using an off-lattice
 * cell population.
 *
 * In cell-centre-based cell populations, each cell is represented by a
 * single node (corresponding to its centre), and connectivity is defined
 * either by a Delaunay triangulation or a radius of influence. In vertex-
 * based cell populations, each cell is represented by a polytope
 * (corresponding to its membrane) with a variable number of vertices.
 * Alternative cell populations may be defined by the user.
 *
 * The OffLatticeSimulation is constructed with a CellPopulation, which
 * updates the correspondence between each Cell and its spatial representation
 * and handles cell division (governed by the CellCycleModel associated
 * with each cell). Once constructed, one or more Force laws may be passed
 * to the OffLatticeSimulation object, to define the mechanical properties
 * of the CellPopulation. Similarly, one or more CellKillers may be passed
 * to the OffLatticeSimulation object to specify conditions in which Cells
 * may die, and one or more CellPopulationBoundaryConditions to specify
 * regions in space beyond which Cells may not move.
```

```
 */
template<unsigned ELEMENT_DIM, unsigned SPACE_DIM = ELEMENT_DIM>
class OffLatticeSimulationPropertyStop : public AbstractCellBasedSimulation<ELEMENT_DIM, SPACE_DIM>
{
private:

    /** Needed for serialization. */
    friend class boost::serialization::access;
    friend class TestOffLatticeSimulation;
    friend class TestOffLatticeSimulationWithNodeBasedCellPopulation;

    /**
     * Save or restore the simulation.
     *
     * @param archive the archive
     * @param version the current version of this class
     */
    template<class Archive>
    void serialize(Archive & archive, const unsigned int version)
    {
        archive & boost::serialization::base_object<AbstractCellBasedSimulation<ELEMENT_DIM, SPACE_DIM> >(*this);
        archive & mForceCollection;
        archive & mBoundaryConditions;
        archive & mpNumericalMethod;
    }

protected:
    boost::shared_ptr<AbstractCellProperty> p_property;
    /** The mechanics used to determine the new location of the cells, a list of the forces. */
    std::vector<boost::shared_ptr<AbstractForce<ELEMENT_DIM, SPACE_DIM> > > mForceCollection;

    /** List of boundary conditions. */
    std::vector<boost::shared_ptr<AbstractCellPopulationBoundaryCondition<ELEMENT_DIM, SPACE_DIM> > > mBoundaryConditions;

    /** The numerical method to use in this simulation. Defaults to the explicit forward Euler method. */
```

```
    boost::shared_ptr<AbstractNumericalMethod<ELEMENT_DIM, SPACE_DIM> > mpNumericalMethod;

    /**
     * Overridden UpdateCellLocationsAndTopology() method.
     *
     * Calculate forces and update node positions.
     */
    virtual void UpdateCellLocationsAndTopology();

    /**
     * Sends nodes back to the positions given in the input map. Used after a failed step
     * when adaptivity is turned on.
     *
     * @param oldNodeLoctions A map linking nodes to their old positions.
     */
    void RevertToOldLocations(std::map<Node<SPACE_DIM>*, c_vector<double, SPACE_DIM> > oldNodeLoctions);

    /**
     * Applies any boundary conditions.
     *
     * @param oldNodeLoctions Mapping between node indices and old node locations
     */
    void ApplyBoundaries(std::map<Node<SPACE_DIM>*, c_vector<double, SPACE_DIM> > oldNodeLoctions);

    /**
     * Overridden SetupSolve() method to clear the forces applied to the nodes.
     */
    virtual void SetupSolve();

    /**
     * Overridden WriteVisualizerSetupFile() method.
     */
    virtual void WriteVisualizerSetupFile();

    bool StoppingEventHasOccurred();

public:
```

```
106
107     /**
108      * Constructor.
109      *
110      * @param rCellPopulation Reference to a cell population object
111      * @param deleteCellPopulationInDestructor Whether to delete the cell
    ↪  population on destruction to
112      *     free up memory (defaults to false)
113      * @param initialiseCells Whether to initialise cells (defaults to
    ↪  true, set to false when loading
114      *     from an archive)
115      */
116     OffLatticeSimulationPropertyStop(AbstractCellPopulation<ELEMENT_DIM,
     ↪  SPACE_DIM>& rCellPopulation,
117                                      bool
                                          ↪  deleteCellPopulationInDestructor
                                          ↪  = false, bool initialiseCells =
                                          ↪  true);
118
119     void SetStopProperty(boost::shared_ptr<AbstractCellProperty>
     ↪  stopPropertySetting);
120
121     bool HasStoppingEventOccurred();
122
123     /**
124      * Add a force to be used in this simulation (use this to set the
    ↪  mechanics system).
125      *
126      * @param pForce pointer to a force law
127      */
128     void AddForce(boost::shared_ptr<AbstractForce<ELEMENT_DIM, SPACE_DIM>
     ↪  > pForce);
129
130     /**
131      * Remove all the forces.
132      */
133     void RemoveAllForces();
134
135     /**
136      * Add a cell population boundary condition to be used in this
    ↪  simulation.
137      *
138      * @param pBoundaryCondition pointer to a boundary condition
```

```
139     */
140    void AddCellPopulationBoundaryCondition(
141
            ↪  boost::shared_ptr<AbstractCellPopulationBoundaryCondition<ELEMENT_DIM
            ↪  SPACE_DIM> > pBoundaryCondition);
142
143    /**
144     * Method to remove all the cell population boundary conditions
145     */
146    void RemoveAllCellPopulationBoundaryConditions();
147
148    /**
149     * Set the numerical method to be used in this simulation (use this
    ↪  to solve the mechanics system).
150     *
151     * @param pNumericalMethod pointer to a numerical method object
152     */
153    void
        ↪  SetNumericalMethod(boost::shared_ptr<AbstractNumericalMethod<ELEMENT_DIM,
        ↪  SPACE_DIM> > pNumericalMethod);
154
155    /**
156     * @return the current numerical method.
157     */
158    const boost::shared_ptr<AbstractNumericalMethod<ELEMENT_DIM,
        ↪  SPACE_DIM> > GetNumericalMethod() const;
159
160    /**
161     * Overridden OutputAdditionalSimulationSetup() method.
162     *
163     * Output any force, boundary condition or numerical method
    ↪  information.
164     *
165     * @param rParamsFile the file stream to which the parameters are
    ↪  output
166     */
167    void OutputAdditionalSimulationSetup(out_stream& rParamsFile);
168
169    /**
170     * Overridden OutputSimulationParameters() method.
171     *
172     * @param rParamsFile the file stream to which the parameters are
    ↪  output
```

```
     */
    virtual void OutputSimulationParameters(out_stream& rParamsFile);

    /**
     * Directly access the forces attached to this simulation, to allow their manipulation after archiving.
     *
     * @return mForceCollection the vector of pointers to forces attached to this simulation
     */
    const std::vector<boost::shared_ptr<AbstractForce<ELEMENT_DIM, SPACE_DIM> > >& rGetForceCollection() const;
};

// Serialization for Boost ≥ 1.36
#include "SerializationExportWrapper.hpp"
EXPORT_TEMPLATE_CLASS_ALL_DIMS(OffLatticeSimulationPropertyStop)

namespace boost
{
namespace serialization
{
/**
 * Serialize information required to construct an OffLatticeSimulation.
 */
template<class Archive, unsigned ELEMENT_DIM, unsigned SPACE_DIM>
inline void save_construct_data(Archive & ar, const OffLatticeSimulationPropertyStop<ELEMENT_DIM, SPACE_DIM> * t,
                                const unsigned int file_version)
{
    // Save data required to construct instance
    const AbstractCellPopulation<ELEMENT_DIM, SPACE_DIM>* p_cell_population = &(t->rGetCellPopulation());
    ar & p_cell_population;
}

/**
 * De-serialize constructor parameters and initialise an OffLatticeSimulation.
 */
template<class Archive, unsigned ELEMENT_DIM, unsigned SPACE_DIM>
inline void load_construct_data(Archive & ar, OffLatticeSimulationPropertyStop<ELEMENT_DIM, SPACE_DIM> * t,
```

```
                                const unsigned int file_version)
{
    // Retrieve data from archive required to construct new instance
    AbstractCellPopulation<ELEMENT_DIM, SPACE_DIM>* p_cell_population;
    ar >> p_cell_population;

    // Invoke inplace constructor to initialise instance, middle two variables set extra
    // member variables to be deleted as they are loaded from archive and to not initialise cells.
    ::new (t) OffLatticeSimulationPropertyStop<ELEMENT_DIM, SPACE_DIM>(*p_cell_population, true, false);
}
}
} // namespace

#endif /*OFFLATTICESIMULATIONPROPERTYSTOP_HPP_*/
```

15.1.31 /src/WanStemCellCycleModel.cpp

```
#include "WanStemCellCycleModel.hpp"

WanStemCellCycleModel::WanStemCellCycleModel() :
        AbstractSimpleCellCycleModel(), mExpandingStemPopulation(false), mPopulation(), mOutput(false), mEventStartTime(
                72.0), mDebug(false), mTimeID(), mVarIDs(), mDebugWriter(), mBasePopulation(), mGammaShift(4.0), mGammaShape(
                2.0), mGammaScale(1.0), mMitoticMode(0), mSeed(0), mTimeDependentCycleDuration(false), mPeakRateTime(), mIncreasingRateSlope(), mDecreasingRateSlope(), mBaseGammaScale(), mHeParamVector(
                { 8, 15, 1, 0, .2, .4, .2, 0, 4, 2, 1, 1 })
{
}

WanStemCellCycleModel::WanStemCellCycleModel(const WanStemCellCycleModel& rModel) :
        AbstractSimpleCellCycleModel(rModel), mExpandingStemPopulation(rModel.mExpandingStemPopulation), mPopulation(
```

```
                rModel.mPopulation), mOutput(rModel.mOutput), mEventStartTime(rModel.mEventStartTime), mDebug(
                rModel.mDebug), mTimeID(rModel.mTimeID), mVarIDs(rModel.mVarIDs), mDebugWriter(rModel.mDebugWriter), mBasePopulation(
                rModel.mBasePopulation), mGammaShift(rModel.mGammaShift), mGammaShape(rModel.mGammaShape), mGammaScale(
                rModel.mGammaScale), mMitoticMode(rModel.mMitoticMode), mSeed(rModel.mSeed), mTimeDependentCycleDuration(
                rModel.mTimeDependentCycleDuration), mPeakRateTime(rModel.mPeakRateTime), mIncreasingRateSlope(
                rModel.mIncreasingRateSlope), mDecreasingRateSlope(rModel.mDecreasingRateSlope), mBaseGammaScale(
                rModel.mBaseGammaScale), mHeParamVector(rModel.mHeParamVector)
{
}

AbstractCellCycleModel* WanStemCellCycleModel::CreateCellCycleModel()
{
    return new WanStemCellCycleModel(*this);
}

void WanStemCellCycleModel::SetCellCycleDuration()
{
    RandomNumberGenerator* p_random_number_generator = RandomNumberGenerator::Instance();

    mCellCycleDuration = mGammaShift + p_random_number_generator->GammaRandomDeviate(mGammaShape, mGammaScale);

    /*************************************
     * CELL CYCLE DURATION RANDOM VARIABLE
     *************************************/
    //Wan stem cell cycle length determined by the same formula as He RPCs
    /*
     if (!mTimeDependentCycleDuration) //Normal operation, cell cycle length stays constant
     {
```

```
41      //He cell cycle length determined by shifted gamma distribution
    ↪   reflecting 4 hr refractory period followed by gamma pdf
42      mCellCycleDuration = mGammaShift +
    ↪   p_random_number_generator->GammaRandomDeviate(mGammaShape,
    ↪   mGammaScale);
43      }*/
44
45     /****
46      * Variable cycle length
47      * Give -ve mIncreasingRateSlope and +ve mDecreasingRateSlope,
48      * cell cycle length linearly declines (increasing rate), then
    ↪   increases, switching at mPeakRateTime
49      ****/
50     /*
51      else
52      {
53      double currTime = SimulationTime::Instance()->GetTime();
54      if (currTime <= mPeakRateTime)
55      {
56      mGammaScale = std::max((mBaseGammaScale - currTime *
    ↪   mIncreasingRateSlope), .0000000000001);
57      }
58      if (currTime > mPeakRateTime)
59      {
60      mGammaScale = std::max(((mBaseGammaScale - mPeakRateTime *
    ↪   mIncreasingRateSlope)
61      + (mBaseGammaScale + (currTime - mPeakRateTime) *
    ↪   mDecreasingRateSlope)),.0000000000001);
62      }
63      Timer::Print("mGammaScale: " + std::to_string(mGammaScale));
64      mCellCycleDuration = mGammaShift +
    ↪   p_random_number_generator->GammaRandomDeviate(mGammaShape,
    ↪   mGammaScale);
65      }
66      */
67 }
68
69 void WanStemCellCycleModel::ResetForDivision()
70 {
71
72     mMitoticMode = 1; //by default, asymmetric division giving rise to He
        ↪   cell (mode 1)
73
```

```
    if (mExpandingStemPopulation)
    {
        double currRetinaAge = SimulationTime::Instance()->GetTime() + mEventStartTime;
        double lensGrowthFactor = .09256 * pow(currRetinaAge, .52728); // power law model fit for lens growth
        unsigned currentPopulationTarget = int(std::round(mBasePopulation * lensGrowthFactor));

        unsigned currentStemPopulation = (mPopulation->GetCellProliferativeTypeCount())[0];
        if (currentStemPopulation < currentPopulationTarget)
        {
            mMitoticMode = 0; //if the current population is < target, symmetrical stem-stem division occurs (mode 0)
        }
    }

    /****************
     * Write mitotic event to relevant files
     * *************/
    if (mDebug)
    {
        WriteDebugData();
    }

    if (mOutput)
    {
        WriteModeEventOutput();
    }

    AbstractSimpleCellCycleModel::ResetForDivision();

}

void WanStemCellCycleModel::Initialise()
{

    boost::shared_ptr<AbstractCellProperty> p_Stem =
                mpCell->rGetCellPropertyCollection().GetCellPropertyRegistry()->Get<S
    mpCell->SetCellProliferativeType(p_Stem);

```

```
    SetCellCycleDuration();
}

void WanStemCellCycleModel::InitialiseDaughterCell()
{

    if (mMitoticMode == 1)
    {
        /*****************************************
         * RPC-fated cells are given HeCellCycleModel
         *****************************************/

        double tiLOffset = -(SimulationTime::Instance()->GetTime());
        //Initialise a HeCellCycleModel and set it up with appropriate TiL value & parameters
        HeCellCycleModel* p_cycle_model = new HeCellCycleModel;
        p_cycle_model->SetModelParameters(tiLOffset, mHeParamVector[0], mHeParamVector[1], mHeParamVector[2],
                                          mHeParamVector[3], mHeParamVector[4], mHeParamVector[5], mHeParamVector[6],
                                          mHeParamVector[7], mHeParamVector[8], mHeParamVector[9], mHeParamVector[10],
                                          mHeParamVector[11]);
        p_cycle_model->EnableKillSpecified();

        //if debug output is enabled for the stem cell, enable it for its progenitor offspring
        if (mDebug)
        {
            p_cycle_model->PassDebugWriter(mDebugWriter, mTimeID, mVarIDs);
        }

        mpCell->SetCellCycleModel(p_cycle_model);
        p_cycle_model->Initialise();
    }

    else
    {
```

```
        SetCellCycleDuration();
    }
}

void WanStemCellCycleModel::SetModelParameters(double gammaShift, double gammaShape, double gammaScale,
                                               std::vector<double> heParamVector)
{
    mGammaShift = gammaShift;
    mGammaShape = gammaShape;
    mGammaScale = gammaScale;
    mHeParamVector = heParamVector;
}

void WanStemCellCycleModel::EnableExpandingStemPopulation(int basePopulation,
                                                          boost::shared_ptr<Abstr
                                                          p_population)
{
    mExpandingStemPopulation = true;
    mBasePopulation = basePopulation;
    mPopulation = p_population;
}

void WanStemCellCycleModel::SetTimeDependentCycleDuration(double peakRateTime, double increasingSlope,
                                                          double decreasingSlope)
{
    mTimeDependentCycleDuration = true;
    mPeakRateTime = peakRateTime;
    mIncreasingRateSlope = increasingSlope;
    mDecreasingRateSlope = decreasingSlope;
    mBaseGammaScale = mGammaScale;
}

void WanStemCellCycleModel::EnableModeEventOutput(double eventStart, unsigned seed)
{
    mOutput = true;
    mEventStartTime = eventStart;
```

```
    mSeed = seed;
}

void WanStemCellCycleModel::WriteModeEventOutput()
{
    double currentTime = SimulationTime::Instance()->GetTime() + mEventStartTime;
    CellPtr currentCell = GetCell();
    double currentCellID = (double) currentCell->GetCellId();
    (*LogFile::Instance()) << currentTime << "\t" << mSeed << "\t" << currentCellID << "\t" << mMitoticMode << "\n"; //
}

void WanStemCellCycleModel::EnableModelDebugOutput(boost::shared_ptr<ColumnDataWriter> debugWriter)
{
    mDebug = true;
    mDebugWriter = debugWriter;

    mTimeID = mDebugWriter->DefineUnlimitedDimension("Time", "h");
    mVarIDs.push_back(mDebugWriter->DefineVariable("CellID", "No"));
    mVarIDs.push_back(mDebugWriter->DefineVariable("TiL", "h"));
    mVarIDs.push_back(mDebugWriter->DefineVariable("CycleDuration", "h"));
    mVarIDs.push_back(mDebugWriter->DefineVariable("Phase2Boundary", "h"));
    mVarIDs.push_back(mDebugWriter->DefineVariable("Phase3Boundary", "h"));
    mVarIDs.push_back(mDebugWriter->DefineVariable("Phase", "No"));
    mVarIDs.push_back(mDebugWriter->DefineVariable("MitoticModeRV", "Percentile"));
    mVarIDs.push_back(mDebugWriter->DefineVariable("MitoticMode", "Mode"));
    mVarIDs.push_back(mDebugWriter->DefineVariable("Label", "binary"));

    mDebugWriter->EndDefineMode();
}

void WanStemCellCycleModel::WriteDebugData()
{
    double currentTime = SimulationTime::Instance()->GetTime();
    double currentCellID = mpCell->GetCellId();
```

```

    mDebugWriter->PutVariable(mTimeID, currentTime);
    mDebugWriter->PutVariable(mVarIDs[0], currentCellID);
    mDebugWriter->PutVariable(mVarIDs[1], 0);
    mDebugWriter->PutVariable(mVarIDs[2], mCellCycleDuration);
    mDebugWriter->PutVariable(mVarIDs[3], 0);
    mDebugWriter->PutVariable(mVarIDs[4], 0);
    mDebugWriter->PutVariable(mVarIDs[5], 0);
    mDebugWriter->PutVariable(mVarIDs[7], mMitoticMode);
    mDebugWriter->AdvanceAlongUnlimitedDimension();
}

/******************
 * UNUSED FUNCTIONS (required for class nonvirtuality, do not remove)
 ******************/

double WanStemCellCycleModel::GetAverageTransitCellCycleTime()
{
    return (0.0);
}

double WanStemCellCycleModel::GetAverageStemCellCycleTime()
{
    return (0.0);
}

void WanStemCellCycleModel::OutputCellCycleModelParameters(out_stream&
 ↪  rParamsFile)
{
    *rParamsFile << "\t\t\t<CellCycleDuration>" << mCellCycleDuration <<
     ↪  "</CellCycleDuration>\n";

// Call method on direct parent class

     ↪  AbstractSimpleCellCycleModel::OutputCellCycleModelParameters(rParamsFile);
}

// Serialization for Boost ≥ 1.36
#include "SerializationExportWrapperForCpp.hpp"
CHASTE_CLASS_EXPORT(WanStemCellCycleModel)
```

15.1.32 /src/WanStemCellCycleModel.hpp

```
#ifndef WANSTEMCELLCYCLEMODEL_HPP_
#define WANSTEMCELLCYCLEMODEL_HPP_

#include "AbstractCellPopulation.hpp"
#include "AbstractSimpleCellCycleModel.hpp"
#include "RandomNumberGenerator.hpp"
#include "Cell.hpp"
#include "StemCellProliferativeType.hpp"
#include "SmartPointers.hpp"
#include "ColumnDataWriter.hpp"
#include "LogFile.hpp"

#include "HeCellCycleModel.hpp"

/***********************************
 * WAN STEM CELL CYCLE MODEL
 * A simple model standing in for the CMZ "stem cells" described in'
 * Wan et al. 2016 [Wan2016]
 *
 * USE: By default, WanStemCellCycleModels consistently divide asymmetrically.
 * InitialiseDaughterCell() marks offspring for RPC fate
 * So-marked cells are given HeCellCycleModels for their next division.
 *
 * Change default model parameters with SetModelParameters(<params>);
 *
 * 2 per-model-event output modes:
 * EnableModeEventOutput() enables mitotic mode event logging-all cells will write to the singleton log file
 * EnableModelDebugOutput() enables more detailed debug output, each seed will have its own file written to
 * by a ColumnDataWriter passed to it from the test
 * (eg. by the SetupDebugOutput helper function in the project simulator)
 *
 * 1 mitotic-event-sequence sampler (only samples one "path" through the lineage):
 * EnableSequenceSampler() - one "sequence" of progenitors writes mitotic event type to a string in the singleton log file
 *
 ***********************************/

```

```
class WanStemCellCycleModel : public AbstractSimpleCellCycleModel
{
    friend class TestSimpleCellCycleModels;

private:

    /** Needed for serialization. */
    friend class boost::serialization::access;
    /**
     * Archive the cell-cycle model and random number generator, never
 ↪  used directly - boost uses this.
     *
     * @param archive the archive
     * @param version the current version of this class
     */
    template<class Archive>
    void serialize(Archive & archive, const unsigned int version)
    {
        archive &
         ↪  boost::serialization::base_object<AbstractSimpleCellCycleModel>(*this);

        SerializableSingleton<RandomNumberGenerator>* p_wrapper =

                 ↪  RandomNumberGenerator::Instance()->GetSerializationWrapper();
        archive & p_wrapper;
        archive & mCellCycleDuration;
    }

    //Private write functions for models
    void WriteModeEventOutput();
    void WriteDebugData();

protected:
    //mode/output variables
    bool mExpandingStemPopulation;
    boost::shared_ptr<AbstractCellPopulation<2>> mPopulation;
    bool mOutput;
    double mEventStartTime;
    //debug writer stuff
    bool mDebug;
    int mTimeID;
    std::vector<int> mVarIDs;
    boost::shared_ptr<ColumnDataWriter> mDebugWriter;
```

```
    //model parameters and state memory vars
    int mBasePopulation;
    double mGammaShift;
    double mGammaShape;
    double mGammaScale;
    unsigned mMitoticMode;
    unsigned mSeed;
    bool mTimeDependentCycleDuration;
    double mPeakRateTime;
    double mIncreasingRateSlope;
    double mDecreasingRateSlope;
    double mBaseGammaScale;
    std::vector<double> mHeParamVector;

    /**
     * Protected copy-constructor for use by CreateCellCycleModel().
     *
     * The only way for external code to create a copy of a cell cycle model
     * is by calling that method, to ensure that a model of the correct subclass is created.
     * This copy-constructor helps subclasses to ensure that all member variables are correctly copied when this happens.
     *
     * This method is called by child classes to set member variables for a daughter cell upon cell division.
     * Note that the parent cell cycle model will have had ResetForDivision() called just before CreateCellCycleModel() is called,
     * so performing an exact copy of the parent is suitable behaviour. Any daughter-cell-specific initialisation
     * can be done in InitialiseDaughterCell().
     *
     * @param rModel the cell cycle model to copy.
     */
    WanStemCellCycleModel(const WanStemCellCycleModel& rModel);

public:

    /**
     * Constructor - just a default, mBirthTime is set in the AbstractCellCycleModel class.
     */
```

```
    WanStemCellCycleModel();

    /**
     * SetCellCycleDuration() method to set length of cell cycle
     */
    void SetCellCycleDuration();

    /**pro
     * Overridden builder method to create new copies of
     * this cell-cycle model.
     *
     * @return new cell-cycle model
     */
    AbstractCellCycleModel* CreateCellCycleModel();

    /**
     * Overridden ResetForDivision() method.
     **/
    void ResetForDivision();

    /**
     * Overridden Initialise() method.
     * Sets proliferative type as stem cell
     **/

    void Initialise();

    /**
     * Overridden InitialiseDaughterCell() method.
     * Used to implement asymmetric mitotic mode
     * Daughter cells are initialised w/ He cell cycle model
     * */
    void InitialiseDaughterCell();

    /*Model setup functions for standard He (SetModelParameters) and
     ↪  deterministic alternative (SetDeterministicMode) models
     * Default parameters are from refits of He et al + deterministic
↪   alternatives
     * He 2012 params: mitoticModePhase2 = 8, mitoticModePhase3 = 15,
↪   p1PP = 1, p1PD = 0, p2PP = .2, p2PD = .4, p3PP = .2, p3PD = 0
     * gammaShift = 4, gammaShape = 2, gammaScale = 1, sisterShift = 1
     */
```

```
    void SetModelParameters(double gammaShift = 4, double gammaShape = 2, double gammaScale = 1, std::vector<double> heParamVector = { 8, 15, 1, 0, .2, .4, .2, 0, 4, 2, 1, 1 });
    void EnableExpandingStemPopulation(int basePopulation, boost::shared_ptr<AbstractCellPopulation<2>> p_population);
    void SetTimeDependentCycleDuration(double peakRateTime, double increasingSlope, double decreasingSlope);

    //Functions to enable per-cell mitotic mode logging for mode rate & sequence sampling fixtures
    //Uses singleton logfile
    void EnableModeEventOutput(double eventStart, unsigned seed);

    //More detailed debug output. Needs a ColumnDataWriter passed to it
    //Only declare ColumnDataWriter directory, filename, etc; do not set up otherwise
    void EnableModelDebugOutput(boost::shared_ptr<ColumnDataWriter> debugWriter);

    //Not used, but must be overwritten lest WanStemCellCycleModels be abstract
    double GetAverageTransitCellCycleTime();
    double GetAverageStemCellCycleTime();

    /**
     * Overridden OutputCellCycleModelParameters() method.
     *
     * @param rParamsFile the file stream to which the parameters are output
     */
    virtual void OutputCellCycleModelParameters(out_stream& rParamsFile);
};

#include "SerializationExportWrapper.hpp"
// Declare identifier for the serializer
CHASTE_CLASS_EXPORT(WanStemCellCycleModel)

#endif /*WANSTEMCELLCYCLEMODEL_HPP_*/
```

15.2 BioBackgroundModels

15.2.1 /README.md

```
## BioBackgroundModels
[![Build
↪  Status](https://travis-ci.org/mmattocks/BioBackgroundModels.jl.svg?branch=master)
[![codecov](https://codecov.io/gh/mmattocks/BioBackgroundModels.jl/branch/master/grap
[![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.repostatu
```

15.2.2 /src/BioBackgroundModels.jl

```
module BioBackgroundModels

import Base:copy,size
import Distances: euclidean
import Distributions:Univariate,Dirichlet,Categorical,logpdf,isprobvec
import Distributed: RemoteChannel, myid, remote_do, rmprocs
import HMMBase: AbstractHMM, assert_hmm, istransmat
import MCMCChains: Chains, ChainDataFrame, heideldiag
import Printf: @sprintf
import Random: rand, shuffle
import Serialization: serialize
import StatsFuns: logsumexp, logaddexp
import Statistics: mean
import UnicodePlots: lineplot,lineplot!, scatterplot,scatterplot!
using BioSequences, DataFrames, FASTX, GFF3, ProgressMeter

include("BHMM/BHMM.jl")
export BHMM
include("EM/chain.jl")
export Chain_ID, EM_step
include("API/genome_sampling.jl")
export setup_sample_jobs, execute_sample_jobs
include("API/EM_master.jl")
export setup_EM_jobs!, execute_EM_jobs!

include("reports/chain_report.jl")
include("reports/partition_report.jl")
include("reports/replicate_convergence.jl")
```

```
include("API/reports.jl")
export generate_reports

include("EM/baum-welch.jl")
include("EM/churbanov.jl")
include("utilities/load_balancer.jl")
export LoadConfig
include("EM/EM_converge.jl")
include("genome_sampling/partition_masker.jl")
include("genome_sampling/sequence_sampler.jl")
include("likelihood_funcs/bg_lh_matrix.jl")
export BGHMM_likelihood_calc
include("likelihood_funcs/hmm.jl")
export obs_lh_given_hmm
include("utilities/observation_coding.jl")
include("utilities/BBG_analysis.jl")
include("utilities/BBG_progressmeter.jl")
include("utilities/HMM_init.jl")
include("utilities/model_display.jl")
include("utilities/utilities.jl")
export split_obs_sets
include("utilities/log_prob_sum.jl")
export lps
end # module
```

15.2.3 /src/API/EM_master.jl

```
#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
#resumes any existing non-converged chains, otherwise initialises hmms for new chains given provided constants
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))
      ↪  #channel to hold BHMM learning jobs
10    output_channel= RemoteChannel(()→Channel{Tuple}(Inf)) #channel to
      ↪  take EM iterates off of
11
12    code_dict = code_job_obs(job_ids, obs_sets)
13
14    @showprogress 1 "Setting up HMMs ..." for id in job_ids #for each
      ↪  jobid, add an initial BHMM to input_channel for EM_workers
15        if haskey(chains, id) && length(chains[id]) > 0 #true if resuming
          ↪  from incomplete chain
16            chain_end=chains[id][end]
17            if !chain_end.converged || (chain_end.converged &&
              ↪  chain_end.delta > delta_thresh)#push the last hmm iterate
              ↪  for nonconverged chains to the input channel with coded
              ↪  observations and values for chain resumption
18                put!(input_channel, (id, chain_end.iterate,
                  ↪  chain_end.hmm, chain_end.log_p, code_dict[(id.obs_id,
                  ↪  id.order)]))
19            else #skip any jobs that have converged from previous runs
20                no_input_hmms -= 1
21            end
22
23        else
24            hmm = init_function(id.K, id.order, id.obs_id) #initialise
              ↪  first BHMM in chain
25            chains[id] = Vector{EM_step}() #initialise the relevant chain
26            obs=code_dict[(id.obs_id,id.order)]
27            lh=obs_lh_given_hmm(obs,hmm,linear=false)
28            put!(input_channel, (id, 1, hmm, lh, obs))
29        end
30    end
31    return no_input_hmms, chains, input_channel, output_channel
32 end
33
34 function execute_EM_jobs!(worker_pool::Vector{Int64},
   ↪  no_input_hmms::Integer, chains::Dict{Chain_ID,Vector{EM_step}},
   ↪  input_channel::RemoteChannel, output_channel::RemoteChannel,
   ↪  chains_path::String; load_dict=Dict{Int64,LoadConfig}(),
   ↪  EM_func::Function=linear_step, delta_thresh=1e-3, max_iterates=5000,
   ↪  verbose=false)
```

```
    #argument checking
    length(worker_pool) < 1 && throw(ArgumentError("Worker pool must contain one or more worker IDs!"))
    no_input_hmms < 1 && throw(ArgumentError("Zero input HMMs reported, likely continuing from chains already converged beyond default delta_thresh for setup_EM_jobs"))
    length(chains) < 1 && throw(ArgumentError("No chains supplied, likely job set from setup_EM_jobs passed incorrectly"))
    !isready(input_channel) && throw(ArgumentError("BHMM input channel has no contents, likely job set from setup_EM_jobs already executed"))

    #SEND BHMM FIT JOBS TO WORKERS
    if isready(input_channel) > 0
        @info "Fitting HMMs.."
        #WORKERS FIT HMMS
        for worker in worker_pool
            if worker in keys(load_dict)
                remote_do(EM_converge!, worker, input_channel, output_channel, no_input_hmms, load_config=load_dict[worker], EM_func=EM_func, delta_thresh=delta_thresh, max_iterates=max_iterates, verbose=verbose)
            else
                remote_do(EM_converge!, worker, input_channel, output_channel, no_input_hmms, EM_func=EM_func, delta_thresh=delta_thresh, max_iterates=max_iterates, verbose=verbose)
            end
        end
    else
        @warn "No input HMMs (all already converged?), skipping fitting.."
    end

    #GET LEARNT HMMS OFF REMOTECHANNEL, SERIALISE AT EVERY ITERATION, UPDATE PROGRESS METERS
    job_counter=no_input_hmms
    learning_meters=Dict{Chain_ID, ProgressHMM}()
    overall_meter=Progress(no_input_hmms,"Overall batch progress:")

    while job_counter > 0
        wait(output_channel)
```

```
        workerid, jobid, iterate, hmm, log_p, delta, converged, steptime = take!(output_channel)
        #either update an existing ProgressHMM meter or create a new one for the job
        if haskey(learning_meters, jobid) && iterate > 2
                update!(learning_meters[jobid], delta, steptime)
        else
            offset = workerid - 1
            if iterate ≤ 2
                learning_meters[jobid] = ProgressHMM(delta_thresh, "$jobid on Wk $workerid:", offset, 2)
                update!(learning_meters[jobid], delta, steptime)
            else
                learning_meters[jobid] = ProgressHMM(delta_thresh, "$jobid on Wk $workerid:", offset, iterate)
                update!(learning_meters[jobid], delta, steptime)
            end
        end
        #push the hmm and related params to the results_dict
        push!(chains[jobid], EM_step(iterate, hmm, log_p, delta, converged))
        #try to serialize the results; catch interrupts and other errors to prevent corruption
        try
            serialize(chains_path, chains)
        catch e
            @warn "Serializing failed!"
            println(e)
        end

        #decrement the job counter, update overall progress meter, and save the current results dict on convergence or max iterate
        if converged || iterate == max_iterates
            job_counter -= 1
            ProgressMeter.update!(overall_meter, (no_input_hmms-job_counter))
            if !isready(input_channel) #if there are no more jobs to be learnt, retire the worker
                workerid!=1 && rmprocs(workerid)
            end
        end
    end

```

```
    #count converged & unconverged jobs, report results
    converged_counter = 0
    unconverged_counter = 0
    for (id, chain) in chains
        chain[end].converged == true ? (converged_counter += 1) : (unconverged_counter += 1)
    end

    @info "Background HMM batch learning task complete, $converged_counter converged jobs, $unconverged_counter jobs failed to converge in $max_iterates iterates since job start."
end
```

15.2.4 /src/API/genome_sampling.jl

```
#function to partition genome and set up Distributed RemoteChannels so partitions can be sampled simultaneously
function setup_sample_jobs(genome_path::String, genome_index_path::String, gff3_path::String, sample_set_length::Integer, sample_window_min::Integer, sample_window_max::Integer, perigenic_pad::Integer; deterministic::Bool=false)
    #argument checking
    !ispath(genome_path) && throw(ArgumentError("Bad genome path!"))
    !ispath(genome_index_path) && throw(ArgumentError("Bad genome index path!"))
    !ispath(gff3_path) && throw(ArgumentError("Bad gff3 path!"))
    sample_set_length < 1 && throw(ArgumentError("Sample set length must be a positive integer!"))
    sample_window_min < 1 || sample_window_max < 1 && throw(ArgumentError("Sample window minimum and maximum bounds must be positive integers!"))
    sample_window_min ≥ sample_window_max && throw(ArgumentError("Sample window minimum size must be smaller than maximum size"))
    perigenic_pad < 0 && throw(ArgumentError("Perigenic pad must be 0 or positive!"))

    coordinate_partitions = partition_genome_coordinates(gff3_path, perigenic_pad)
    sample_sets = DataFrame[]
```

```
    input_sample_jobs = RemoteChannel(()→Channel{Tuple}(length(coordinate_partitions))) #channel to hold sampling jobs
    completed_sample_jobs = RemoteChannel(()→Channel{Tuple}(length(coordinate_partitions))) #channel to hold completed sample dfs
    for (partitionid, partition) in coordinate_partitions
        add_metacoordinates!(partition)
        put!(input_sample_jobs, (genome_path, genome_index_path, partition, partitionid, sample_set_length, sample_window_min, sample_window_max, deterministic))
    end
    progress_channel = RemoteChannel(()→Channel{Tuple}(20))
    return (input_sample_jobs, completed_sample_jobs, progress_channel, sample_set_length)
end

function execute_sample_jobs(channels::Tuple{RemoteChannel,RemoteChannel,RemoteChannel,Int
worker_pool::Vector{Int64}; partitions::Integer=3)
    input_sample_channel, completed_sample_channel, progress_channel, sample_set_length = channels

    #argument checking
    length(worker_pool) < 1 && throw(ArgumentError("Worker pool must contain one or more worker IDs!"))
    !isready(input_sample_channel) && throw(ArgumentError("Input sample channel not ready, likely channel set from setup_sample_jobs passed incorrectly"))
    sample_set_length < 1 && throw(ArgumentError("Sample set length must be a positive integer, likely channel set from setup_sample_jobs passed incorrectly"))

    #send sampling jobs to workers
    if isready(input_sample_channel) > 0
        @info "Sampling.."
        #WORKERS SAMPLE
        for (n, worker) in enumerate(worker_pool)
            if n ≤ partitions #no more workers than partitions to be used
                remote_do(get_sample_set, worker, input_sample_channel, completed_sample_channel, progress_channel)
            end
```

```
        end
    else
        @error "No sampling jobs!"
    end

    #progress meters for sampling
    sampling_meters=Dict{String, Progress}()
    overall_sampling_meter=Progress(partitions,"Overall sampling progress:")
    completed_counter = 0
    ProgressMeter.update!(overall_sampling_meter, completed_counter)
    sampling_offset = ones(Bool, partitions)

    #collect progress updates while waiting on completion of sampling jobs
    while completed_counter < partitions
        wait(progress_channel)
        partition_id, progress = take!(progress_channel)
        if haskey(sampling_meters, partition_id)
            ProgressMeter.update!(sampling_meters[partition_id], progress)
        else
            offset = findfirst(sampling_offset)[1]
            sampling_meters[partition_id] = Progress(sample_set_length, "Sampling partition $partition_id:", offset)
            ProgressMeter.update!(sampling_meters[partition_id], progress)
            sampling_offset[offset] = false
        end
        if progress == sample_set_length
            completed_counter += 1
            ProgressMeter.update!(overall_sampling_meter, completed_counter)
        end
    end

    #collect sample dfs by partition id when ready
    sample_record_dfs = Dict{String,DataFrame}()
    collected_counter = 0
    while collected_counter < partitions
        wait(completed_sample_channel)
        partition_id, sample_df = take!(completed_sample_channel)
        sample_record_dfs[partition_id] = sample_df
```

```
        collected_counter += 1
        @info "Partition $partition_id completed sampling..."
    end

    return sample_record_dfs
end
```

15.2.5 /src/API/reports.jl

```
struct Report_Folder
    partition_id::String
    partition_report::Partition_Report
    replicate_report::Replicate_Report
    chain_reports::Dict{Chain_ID,Chain_Report}
end

function generate_reports(chains::Dict{Chain_ID,Vector{EM_step}},
↪   test_sets::Dict{String,Vector{LongSequence{DNAAlphabet{2}}}})
    #check chains dict for problems

    length(chains)==0 && throw(ArgumentError("Empty chains dict!"))
    any(chain→length(chain)<2, values(chains)) &&
     ↪  throw(ArgumentError("Some chains are too short (<2 EM steps)!
     ↪  Probably not all chains have been operated on by EM workers yet.
     ↪  Try EM_converge first!"))
    unconv=sum(!chain[end].converged for chain in values(chains))
    unconv > 0 && @warn "Not all chains are converged to the selected
     ↪  step delta."
    length(test_sets)==0 && throw(ArgumentError("Empty test_sets dict!"))

    chain_reports = report_chains(chains, test_sets)
    partition_reports = report_partitions(chain_reports)

    report_folders=Dict{String, Report_Folder}()

    for part_report in partition_reports
        rep_report=report_replicates(part_report.best_repset, chains)
        chain_subset=Dict{Chain_ID,Chain_Report}()
        for id in keys(chain_reports)
            id.obs_id==part_report.partition_id &&
             ↪  (chain_subset[id]=chain_reports[id])
        end
```

```
        ↪ report_folders[part_report.partition_id]=Report_Folder(part_report.partit
        ↪ part_report, rep_report, chain_subset)
    end

    return report_folders
end
```

15.2.6 /src/BHMM/BHMM.jl

```
"""
    BHMM([a::AbstractVector{T}, ]A::AbstractMatrix{T},
↪   B::AbstractVector{<:Categorical}) where F where T
Build an BHMM with transition matrix `A` and observations distributions
↪   `B`.
If the initial state distribution `a` is not specified, a uniform
↪   distribution is assumed.
Observations distributions can be of different types (for example
↪   `Normal` and `Exponential`).
However they must be of the same dimension (all scalars or all
↪   multivariates).
# Example
```julia
hmm = BHMM([0.9 0.1; 0.1 0.9], [Normal(0,1), Normal(10,1)])
```
"""
struct BHMM{T} <: AbstractHMM{Univariate}
    a::AbstractVector{T}
    A::AbstractMatrix{T}
    B::AbstractVector{<:Categorical}
    partition::String
    BHMM{T}(a, A, B, partition="") where {T} = assert_BHMM(a, A, B) &&
     ↪ new(a, A, B, partition)
end

BHMM(a::AbstractVector{T}, A::AbstractMatrix{T},
↪   B::AbstractVector{<:Categorical}, partition ...) where {T} =
↪   BHMM{T}(a, A, B, partition ...)
BHMM(A::AbstractMatrix{T}, B::AbstractVector{<:Categorical},
↪   partition ...) where {T} = BHMM{T}(ones(size(A)[1])/size(A)[1], A, B,
↪   partition ...)
```

```
copy(hmm::BHMM) = BHMM(copy(hmm.a), copy(hmm.A), copy(hmm.B), hmm.partition)
size(hmm::BHMM) = (length(hmm.B), length(hmm.B[1].p))


"""
    assert_BHMM(a, A, B)
Throw an `ArgumentError` if the initial state distribution and the transition matrix rows does not sum to 1,
and if the observations distributions does not have the same dimensions.
"""
function assert_BHMM(a::AbstractVector,
                     A::AbstractMatrix,
                     B::AbstractVector{<:Categorical})
    !isprobvec(a) && throw(ArgumentError("Initial state vector a is not a valid probability vector!"))
    !istransmat(A) && throw(ArgumentError("Transition matrix A is not valid!"))

    !all([length(d.p) for d in B] .== length(B[1].p)) && throw(ArgumentError("All distributions must have the same dimensions"))
    !(length(a) == size(A,1) == length(B)) && throw(ArgumentError("Length of initial state vector a, dimension of transition matrix A, and number of distributions B are not the same!"))
    return true
end

function Base.show(io::IO, hmm::BHMM)
    println(io, "Background BHMM")
    println(io, "State Initial and Transition Probabilities")
    print(io, "a: ")
    show(io, hmm.a)
    println(io)
    print(io, "A: ")
    display(hmm.A)
    println(io)
    println(io, "INFORMATIVE SYMBOLS BY STATE")
    for (n,d) in enumerate(hmm.B)
        print(io, "K$n ")
        print_emitters(d)
    end
end
```

15.2.7 /src/EM/EM_converge.jl

```
function EM_converge!(hmm_jobs::RemoteChannel, output_hmms::RemoteChannel, no_models::Integer; load_config::LoadConfig=LoadConfig(1:typemax(Int64)-1, 0:typemax(Int64)-1),  EM_func::Function=linear_step, delta_thresh=1e-3, max_iterates=5000, verbose=false)
    while isready(hmm_jobs)
        workerid = myid()
        jobid, start_iterate, hmm, job_norm, observations = load_balancer(no_models, hmm_jobs, load_config)
        jobid == 0 && break #no valid job for this worker according to load_table entry

        start_iterate > max_iterates - 1 && throw(ArgumentError("BHMM chain $jobid is already longer ($start_iterate iterates) than specified max_iterates!"))
        curr_iterate = start_iterate

        #mask calculations here rather than ms_mle_step to prevent recalculation every iterate
        #build array of observation lengths

        obs_lengths = [findfirst(iszero,observations[o,:])-1 for o in 1:size(observations,1)] #mask calculations here rather than mle_step to prevent recalculation every iterate
        EM_func==bw_step && (observations=transpose(observations))

        start_iterate == 1 && put!(output_hmms, (workerid, jobid, curr_iterate, hmm, job_norm, 0.0, false, 0.0)); #on the first iterate return the initial BHMM immediately
        verbose && @info "Fitting BHMM on Wk $workerid, start iterate $start_iterate, $jobid with $(size(hmm)[1]) states and $(size(hmm)[2]) symbols ... "

        curr_iterate += 1

        start=time()
        new_hmm, last_norm = EM_func(hmm, observations, obs_lengths)
        delta = abs(lps(job_norm, -last_norm))
```

```
        put!(output_hmms, (workerid, jobid, curr_iterate, new_hmm, last_norm, delta, false, time()-start))

        for i in curr_iterate:max_iterates
            start=time()
            new_hmm, norm = EM_func(new_hmm, observations, obs_lengths)
            curr_iterate += 1
            delta = abs(lps(norm, -last_norm))
            if delta < delta_thresh
                put!(output_hmms, (workerid, jobid, curr_iterate, new_hmm, norm, delta, true, time()-start))
                verbose && @info "$jobid converged after $(curr_iterate-1) EM steps"
                break
            else
                put!(output_hmms, (workerid, jobid, curr_iterate, new_hmm, norm, delta, false, time()-start))
                verbose && @info "$jobid EM step $(curr_iterate-1) delta $delta"
                last_norm = norm
            end
        end
    end
end
```

15.2.8 /src/EM/baum-welch.jl

```
"""
    ms_mle_step(hmm::AbstractHMM{F}, observations) where F

Perform one step of the EM (Baum-Welch) algorithm.

# Example
```julia
hmm, log_likelihood = ms_mle_step(hmm, observations)
```
"""

function bw_step(hmm::BHMM, observations, obs_lengths)
    lls = bw_llhs(hmm, observations)
    log_α = messages_forwards_log(hmm.a, hmm.A, lls, obs_lengths)
    log_β = messages_backwards_log(hmm.A, lls, obs_lengths)
```

```
    log_A = log.(hmm.A)

    K,Tmaxplus1,O = size(lls) #the last T value is the 0 end marker of the longest T

    #transforms to cut down log_ξ, log_γ assignment times
    lls = permutedims(lls, [2,3,1]) # from (K,T,O) to (T,O,K)
    log_α = permutedims(log_α, [2,3,1])
    log_β = permutedims(log_β, [2,3,1])

    # E-step
    log_ξ = fill(-Inf, Tmaxplus1,O,K,K)
    log_γ = fill(-Inf, Tmaxplus1,O,K)
    log_pobs = zeros(O)

   @inbounds for o = 1:O
        log_pobs[o] = logsumexp(lps.(log_α[1,o,:], log_β[1,o,:]))
    end

    Threads.@threads for idx in [(i,j,o) for i=1:K, j=1:K, o=1:O]
        i,j,o = idx[1],idx[2],idx[3]
        obsl = obs_lengths[o]
        @inbounds for t = 1:obsl-1 #log_ξ & log_γ calculated to T-1 for each o
           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])
           log_γ[t,o,i] = lps(log_α[t,o,i], log_β[t,o,i], -log_pobs[o])
        end
        t=obsl #log_ξ @ T = 0
        log_ξ[t,o,i,j] = 0
        log_γ[t,o,i] = lps(log_α[t,o,i], log_β[t,o,i], -log_pobs[o])
    end

    ξ = similar(log_ξ)
    ξ .= exp.(log_ξ)
    Σk_ξ = sum(ξ, dims=[3,4])
    nan_mask = Σk_ξ .== 0
    Σk_ξ[nan_mask] .= Inf #prevent NaNs in dummy renorm
    ξ  ./=  Σk_ξ #dummy renorm across K to keep numerical creep from causing isprobvec to fail on new new_A during hmm creation

    γ = similar(log_γ)
    γ .= exp.(log_γ)
```

```
    Σk_γ = sum(γ, dims=3)
    Σk_γ[nan_mask[:,:,:]] ⋅= Inf #prevent NaNs in dummy renorm
    γ ./= Σk_γ #dummy renorm

    # M-step
    new_A = zeros(K,K)
    for i=1:K, j=1:K
        Σotξ_vec = zeros(O)
        Σotγ_vec = zeros(O)
        Threads.@threads for o in 1:O
            Σotξ_vec[o] = sum(ξ[1:obs_lengths[o]-1,o,i,j])
            Σotγ_vec[o] = sum(γ[1:obs_lengths[o]-1,o,i])
        end
        new_A[i,j] = sum(Σotξ_vec) / sum(Σotγ_vec)
    end
    new_A ./= sum(new_A, dims=2) #dummy renorm
    new_a = (sum(γ[1,:,:], dims=1)./sum(sum(γ[1,:,:], dims=1)))[1,:]
    new_a ./= sum(new_a) #dummy renorm

    obs_mask = .!nan_mask
    obs_collection = observations[obs_mask[:,:]]

    B = Categorical[]
    @inbounds for (i, d) in enumerate(hmm.B)
        γ_d = γ[:,:,i]
        push!(B, t_categorical_mle(Categorical, d.support[end], obs_collection, γ_d[obs_mask[:,:]]))
    end

    return typeof(hmm)(new_a, new_A, B), lps(log_pobs)
end
                function bw_llhs(hmm, observations)
                    lls = zeros(length(hmm.B), size(observations)...)
                    Threads.@threads for d in 1:length(hmm.B)
                        lls[d,:,:] = logpdf.(hmm.B[d], observations)
                    end
                    return lls
                end
                #Multisequence competent log implementations of forward and backwards algos
                function messages_forwards_log(init_distn, trans_matrix, log_likelihoods, obs_lengths)
                    log_alphas = zeros(size(log_likelihoods))
```

```
    log_trans_matrix = log.(trans_matrix)
    log_alphas[:,1,:] = log.(init_distn) .+ log_likelihoods[:,1,:]
    Threads.@threads for o in 1:size(log_likelihoods)[3]
        @inbounds for t in 2:obs_lengths[o], j in 1:size(log_likelihoods)[1]
            log_alphas[j,t,o]=logsumexp(log_alphas[:,t-1,o] .+ log_trans_matrix[:,j]) + log_likelihoods[j,t,o]
        end
    end
    return log_alphas
end

function messages_backwards_log(trans_matrix, log_likelihoods, obs_lengths)
    log_betas = zeros(size(log_likelihoods))
    log_trans_matrix = log.(trans_matrix)
    Threads.@threads for o in 1:size(log_likelihoods)[3]
        @inbounds for t in obs_lengths[o]-1:-1:1
            tmp = view(log_betas, :, t+1, o) .+ view(log_likelihoods, :, t+1, o)
            @inbounds for i in 1:size(log_likelihoods)[1]
                log_betas[i,t,o] = logsumexp(view(log_trans_matrix, i,:) .+ tmp)
            end
        end
    end
    return log_betas
end
#subfunc derived from Distributions.jl categorical.jl fit_mle, threaded
function t_categorical_mle(::Type{<:Categorical}, k::Integer, x::AbstractArray{T}, w::AbstractArray{F}) where T<:Integer where F<:AbstractFloat
    Categorical(t_pnormalize!(t_add_categorical_counts!(zeros(k), x, w)))
end

t_pnormalize!(v::Vector) = (v .⁄= sum(v); v)
```

```julia

function t_add_categorical_counts!(h::Vector{F}, x::AbstractArray{T}, w::AbstractArray{F}) where T<:Integer where F<:AbstractFloat
    n = length(x)
    if n ≠ length(w)
        throw(DimensionMismatch("Inconsistent array lengths."))
    end
    hlock = ReentrantLock()
    Threads.@threads for i=1:n
        @inbounds xi = x[i]
        @inbounds wi = w[i]
        lock(hlock)
        h[xi] += wi   # cannot use @inbounds, as no guarantee that x[i] is in bound
        unlock(hlock)
    end
    return h
end
```

15.2.9 /src/EM/chain.jl

```julia
struct Chain_ID
    obs_id::AbstractString
    K::Integer
    order::Integer
    replicate::Integer
    Chain_ID(obs_id,K,order,replicate) = assert_chain_id(K,order,replicate) && new(obs_id,K,order,replicate)
end

function assert_chain_id(K, order, replicate)
    K < 1 && throw(ArgumentError("Chain_ID K (# of hmm states) must be a positive integer!"))
    order < 0 && throw(ArgumentError("Chain_ID symbol order must be zero or a positive integer!"))
    replicate < 1 && throw(ArgumentError("Chain_ID replicate must be a positive integer!"))
    return true
end
```

```julia

struct EM_step
    iterate::Integer
    hmm::BHMM
    log_p::AbstractFloat
    delta::AbstractFloat
    converged::Bool

    EM_step(iterate,hmm,log_p,delta,converged)=assert_step(iterate,hmm,log_p) && new(iterate, hmm, log_p, delta, converged)
end

function assert_step(iterate, hmm, log_p)
    iterate < 1 && throw(ArgumentError("EM_step iterate number must be a positive integer!"))
    assert_hmm(hmm.a, hmm.A, hmm.B)
    log_p > 0.0 && throw(ArgumentError("EM_step log probability value must be 0 or negative!"))
    return true
end
```

15.2.10 /src/EM/churbanov.jl

```julia
function linear_step(hmm, observations, obs_lengths)
    O,T = size(observations);
    a = transpose(log.(hmm.a)); A = log.(hmm.A) #less expensive to transpose transmatrix in the backwards_sweep than to transpose here and take a ranged view ie view(A,m:m,:) is more expensive than transpose(view(A,m,:))
    N = length(hmm.B); Γ = length(hmm.B[1].support);
    mask=observations.≠0
    #INITIALIZATION
    βoi_T = zeros(O,N); βoi_t = zeros(O,N) #log betas at T initialised as zeros
    Eoγim_T = fill(-Inf,O,Γ,N,N); Eoγim_t = fill(-Inf,O,Γ,N,N)
    for m in 1:N, i in 1:N, γ in 1:Γ, o in 1:O
        observations[o, obs_lengths[o]] == γ && m == i && (Eoγim_T[o, γ, i, m] = 0)
    end
    Tijm_T = fill(-Inf,O,N,N,N); Tijm_t = fill(-Inf,O,N,N,N) #Ti,j(T,m) = 0 for all m; in logspace

```

```
    #RECURRENCE

    βoi_T,Tijm_T,Eoγim_T=backwards_sweep!(hmm,A,N,Γ,βoi_T,βoi_t,Tijm_T,Tijm_t,Eoγ

    #TERMINATION
    lls = c_llhs(hmm,observations[:,1])
    α1om = lps.(lls,a) #first position forward msgs
    log_pobs = [c_lse(lps.(α1om[o,:], βoi_T[o,:])) for o in 1:O]

    Toij = [c_lse([lps(Tijm_T[o,i,j,m], α1om[o,m], -log_pobs[o]) for m in 1:N]) for o in 1:O, i in 1:N, j in 1:N] #terminate Tijs with forward messages
    Eoiγ = [c_lse([lps(Eoγim_T[o,γ,i,m], α1om[o,m], -log_pobs[o]) for m in 1:N]) for o in 1:O, i in 1:N, γ in 1:Γ] #terminate Eids with forward messages

    #INTEGRATE ACROSS OBSERVATIONS AND SOLVE FOR NEW BHMM PARAMS
    #INITIAL STATE DIST
    a_o=α1om.+βoi_T•-c_lse.(eachrow(α1om.+βoi_T)) #estimate a for each o
    obs_penalty=log(O) #broadcast subtraction to normalise log prob vals by obs number
    new_a=c_lse.(eachcol(a_o))•-obs_penalty #sum over obs and normalise by number of obs
    #TRANSITION MATRIX
    new_A = [c_lse(view(Toij,:,i,j)) for i in 1:N, j in 1:N]•-c_lse.(eachcol(c_lse.([view(Toij,o,i,:) for o in 1:O, i in 1:N])))
    #EMISSION MATRIX
    new_b=[c_lse(view(Eoiγ,:,i,γ)) for i in 1:N, γ in 1:Γ]•-c_lse.(eachcol(c_lse.([view(Eoiγ,o,i,:) for o in 1:O, i in 1:N])))
    new_B=[Categorical(exp.(new_b[i,:])) for i in 1:N]

    return BHMM(exp.(new_a), exp.(new_A), new_B, hmm.partition), lps(log_pobs)
end
                #LINEAR_STEP SUBFUNCS
                function backwards_sweep!(hmm, A, N, Γ, βoi_T, βoi_t, Tijm_T, Tijm_t, Eoγim_T, Eoγim_t, observations, mask, obs_lengths)
                    for t in maximum(obs_lengths)-1:-1:1
                        last_β=copy(βoi_T)
                        lls = c_llhs(hmm,observations[:,t+1])
```

```
                        omask = findall(mask[:,t+1])
                        βoi_T[omask,:] .+= view(lls,omask,:)
                        Threads.@threads for m in 1:N
                            βoi_t[omask,m] = c_lse.(eachrow(view(βoi_T,omask,:).+transpose(view(A,m
                            for j in 1:N, i in 1:N
                                Tijm_t[omask, i, j, m] •= c_lse.(eachrow(lps.(view(Tijm_T,omask,i,j,:),
                                view(lls,omask,:),
                                transpose(view(A,m,:)))))
                                i==m && (Tijm_t[omask, i, j, m] •= logaddexp.(Tijm_t[omask, i, j, m],
                                lps.(last_β[omask,j],
                                A[m,j],lls[omask,j])))
                            end
                            for i in 1:N, γ in 1:Γ
                                Eoγim_t[omask, γ, i, m] •= c_lse.(eachrow(lps.(view(Eoγim_T,omask,γ,i,:),vie
                                if i==m
                                    symmask = findall(observations[:,t].==γ)
                                    Eoγim_t[symmask, γ, i, m] •= logaddexp.(Eoγim_t[symmask, γ, i,
                                    m], βoi_t[symmask,m])
                                end
                            end
                        end
                        βoi_T=copy(βoi_t); Tijm_T=copy(Tijm_t); Eoγim_T = copy(Eoγim_t);
                    end
                    return βoi_T, Tijm_T, Eoγim_T
                end

                #logsumexp
                function c_lse(X::AbstractArray{T}; dims=:) where {T<:Real}
                    u=maximum(X)
                    u isa AbstractArray || isfinite(u) || return float(u)
                    return u + log(sum(x -> exp(x-u), X))
                end

                #log likelihoods
                function c_llhs(hmm, observation)
```

```
                    lls = zeros(length(observation),length(hmm.B))
                    Threads.@threads for d in 1:length(hmm.B)
                        lls[:,d] = logpdf.(hmm.B[d], observation)
                    end
                    return lls
                end
```

15.2.11 /src/genome_sampling/partition_masker.jl

```
function get_partition_code_dict(dict_forward::Bool=true)
    if dict_forward == true
        return partition_code_dict = Dict("intergenic"⇒1,"periexonic"⇒2,"exon"⇒3)
    else
        return code_partition_dict = Dict(1⇒"intergenic", 2⇒"periexonic", 3⇒"exon")
    end
end

function make_padded_df(position_fasta::String, gff3_path::String, genome_path::String, genome_index_path::String, pad::Integer)
    position_reader = FASTA.Reader(open((position_fasta),"r"))
    genome_reader = open(FASTA.Reader, genome_path, index=genome_index_path)
    scaffold_df = build_scaffold_df(gff3_path)
    position_df = DataFrame(SeqID = String[], Start=Int[], End=Int[], PadSeq = LongSequence[], PadStart=Int[], RelStart=Int[], SeqOffset=Int[])
    scaffold_seq_dict = build_scaffold_seq_dict(genome_path, genome_index_path)

    for entry in position_reader
        scaffold = FASTA.identifier(entry)

        if scaffold ≠ "MT"
            desc_array = split(FASTA.description(entry))
            pos_start = parse(Int, desc_array[2])
            pos_end = parse(Int, desc_array[4])
            scaffold_end = scaffold_df.End[findfirst(isequal(scaffold), scaffold_df.SeqID)]

            pad_start=max(1,pos_start-pad)
```

```
            pad_length= pos_start - pad_start
            seq_offset = pad - pad_length
            padded_seq = fetch_sequence(scaffold, scaffold_seq_dict, pad_start, pos_end, '+')

            if !hasambiguity(padded_seq)
                push!(position_df, [scaffold, pos_start, pos_end, padded_seq, pad_start, pad_length, seq_offset])
            end
        end
    end

    close(position_reader)
    close(genome_reader)
    return position_df
end

function add_partition_masks!(position_df::DataFrame, gff3_path::String, perigenic_pad::Integer=500, columns::Tuple{Symbol,Symbol,Symbol}=(:SeqID, :PadSeq, :PadStart))
    partitions=["exon", "periexonic", "intergenic"]
    partition_coords_dict = partition_genome_coordinates(gff3_path, perigenic_pad)
    partitioned_scaffolds = divide_partitions_by_scaffold(partition_coords_dict)
    maskcol = [zeros(Int64,0,0) for i in 1:size(position_df,1)]
    position_df.MaskMatrix=maskcol

    @Threads.threads for entry in eachrow(position_df)
        scaffold = entry[columns[1]]
        maskLength = length(entry[columns[2]])
        seqStart = entry[columns[3]]

        scaffold_coords_dict = Dict{String,DataFrame}()

        for ((partition, part_scaffold), df) in partitioned_scaffolds
            if scaffold == part_scaffold
                scaffold_coords_dict[partition] = df
            end
        end

        entry.MaskMatrix=mask_sequence_by_partition(maskLength, seqStart, scaffold_coords_dict)
```

```
    end

end
                #add_partition_masks!() SUBFUNCTIONS
                function divide_partitions_by_scaffold(partition_coords_dict::Dict{String, DataFrame})
                    scaffold_coords_dict = Dict{Tuple{String, String}, DataFrame}()
                    for (partition_id, partition_df) in partition_coords_dict
                        for scaffold_subframe in groupby(partition_df, :SeqID)
                            scaffold_id = scaffold_subframe.SeqID[1]
                            scaffold_df = copy(scaffold_subframe)
                            scaffold_coords_dict[(partition_id, scaffold_id)] = scaffold_df
                        end
                    end
                    return scaffold_coords_dict
                end

                function mask_sequence_by_partition(maskLength::Integer, seqStart::Integer, scaffold_coords_dict::Dict{String, DataFrame})
                    partition_code_dict = get_partition_code_dict()
                    seqMask = zeros(Integer, (maskLength, 2))
                    position = seqStart
                    while position ≤ seqStart+maskLength
                        position_partition, partition_extent, position_strand = find_position_partition(position, scaffold_coords_dict)

                        partition_code = partition_code_dict[position_partition]
                        mask_position = position - seqStart + 1

                        seqMask[mask_position:min(maskLength,mask_position + partition_extent),1] .= partition_code
                        if position_strand == '+'
```

```
                                seqMask[mask_position:min(maskLength,mask_position + partition_extent),2] .= 1
                            elseif position_strand == '-'
                                seqMask[mask_position:min(maskLength,mask_position + partition_extent),2] .= -1
                            else
                                seqMask[mask_position:min(maskLength,mask_position + partition_extent),2] .= 0
                            end

                            position += partition_extent + 1
                        end

                        return seqMask
                    end

                    function find_position_partition(position::Integer, partition_dict::Dict{String, DataFrame})
                        foundPos = false
                        position_partition_id = ""
                        three_prime_extent = 0
                        sample_strand = 0
                        for (partition_id, partition) in partition_dict
                            hitindex = findfirst(x->x≥position, partition.End)
                            if hitindex ≠ nothing
                                if position ≥ partition.Start[hitindex]
                                    foundPos=true
                                    position_partition_id = partition_id
                                    three_prime_extent = partition.End[hitindex] - position
                                    if partition_id == "exon" || partition_id == "periexonic"
                                        sample_strand = partition.Strand[hitindex]
                                    end
                                end
                            end
                        end

```

```
                    if foundPos == false
                        throw(DomainError("Position $position not found among partition coordinates!"))
                    else
                        return position_partition_id, three_prime_extent, sample_strand
                    end
                end

#function to partition a genome into coordinate sets of:
#merged exons
#"periexonic" sequences (genes with 5' and 3' boundaries projected -/+perigenic_pad bp, minus exons) - includes promoter elements, introns, 3' elements
#intergenic sequences (everything else)
#given a valid gff3
function partition_genome_coordinates(gff3_path::String, perigenic_pad::Integer=500)
    # construct dataframes of scaffolds and metacoordinates
    scaffold_df = build_scaffold_df(gff3_path)

    #partition genome into intragenic, periexonic, and exonic coordinate sets
    #assemble exonic featureset
    exon_df = DataFrame(SeqID = String[], Start = Integer[], End = Integer[], Strand=Char[])
    build_feature_df!(gff3_path, "CDS", "MT", exon_df)

    #project exon coordinates onto the scaffold bitwise, merging overlapping features and returning the merged dataframe
    merged_exon_df = DataFrame(SeqID = String[], Start = Integer[], End = Integer[], Strand=Char[])
    @showprogress 1 "Partitioning exons..." for scaffold_subframe in DataFrames.groupby(exon_df, :SeqID) # for each scaffold subframe that has exon features
        scaffold_id  = scaffold_subframe.SeqID[1] #get the scaffold id
        scaffold_bitarray = init_scaffold_bitarray(scaffold_id, scaffold_df, false, stranded=true) #init a stranded bitarray of scaffold length

        project_features_to_bitarray!(scaffold_subframe,scaffold_bitarray) #project features as Trues on bitarray of falses
```

```
        merged_subframe = get_feature_df_from_bitarray(scaffold_id, scaffold_bitarray) #get a feature df from the projected bitarray
        append!(merged_exon_df,merged_subframe) #append the merged scaffold df to the overall merged df
    end

    #assemble gene featureset
    gene_df =  DataFrame(SeqID = String[], Start = Integer[], End = Integer[], Strand = Char[])
    build_feature_df!(gff3_path, "gene", "MT", gene_df)

    perigenic_pad > 0 && add_pad_to_coordinates!(gene_df, scaffold_df, perigenic_pad) #if a perigenic pad is specified (to capture promoter/downstream elements etc in the periexonic set), apply it to the gene coords

    #build intergenic coordinate set by subtracting gene features from scaffold bitarrays
    intergenic_df = DataFrame(SeqID = String[], Start = Integer[], End = Integer[])
    @showprogress 1 "Partitioning intergenic regions..." for scaffold_subframe in DataFrames.groupby(scaffold_df, :SeqID) # for each scaffold subframe that has exon features
        scaffold_id  = scaffold_subframe.SeqID[1] #get the scaffold id
        scaffold_bitarray = init_scaffold_bitarray(scaffold_id, scaffold_df, true) #init a bitarray of scaffold length
        if any(isequal(scaffold_id),gene_df.SeqID) #if any genes on the scafold
            scaffold_genes=gene_df[findall(isequal(scaffold_id), gene_df.SeqID), :] #get the gene rows by finding the scaffold_id

            subtract_features_from_bitarray!(scaffold_genes,scaffold_bitarray) #subtract the gene positions from the scaffold bitarray
        end
        intragenic_subframe = get_feature_df_from_bitarray(scaffold_id, scaffold_bitarray) #get a feature df from the projected bitarray
        append!(intergenic_df,intragenic_subframe) #append the merged scaffold df to the overall merged df
    end

```

```
    #build periexonic set by projecting gene coordinates onto the scaffold bitwise, then subtracting the exons
    periexonic_df = DataFrame(SeqID = String[], Start = Integer[], End = Integer[], Strand=Char[])
    @showprogress 1 "Partitioning periexonic regions..." for gene_subframe in DataFrames.groupby(gene_df, :SeqID) # for each scaffold subframe that has exon features
        scaffold_id  = gene_subframe.SeqID[1] #get the scaffold id
        scaffold_bitarray = init_scaffold_bitarray(scaffold_id, scaffold_df, false, stranded=true) #init a bitarray of scaffold length
        project_features_to_bitarray!(gene_subframe,scaffold_bitarray) #project features as Trues on bitarray of falses
        if any(isequal(scaffold_id),merged_exon_df.SeqID)
            scaffold_exons=merged_exon_df[findall(isequal(scaffold_id), merged_exon_df.SeqID), :]

            subtract_features_from_bitarray!(scaffold_exons,scaffold_bitarray)
        end
        periexonic_subframe = get_feature_df_from_bitarray(scaffold_id, scaffold_bitarray) #get a feature df from the projected bitarray
        append!(periexonic_df,periexonic_subframe) #append the merged scaffold df to the overall merged df
    end

    return Dict("exon"⇒merged_exon_df, "periexonic"⇒periexonic_df, "intergenic"⇒intergenic_df)
end

            #partition_genome_coordinates() SUBFUNCTIONS
            #BITARRAY SCAFFOLD REPRESENTATION SUBFUNCTIONS
            function init_scaffold_bitarray(scaffold_id::String, scaffold_df, value::Bool; stranded::Bool=false)
                scaffold_length = scaffold_df.End[findfirst(isequal(scaffold_id), scaffold_df.SeqID)]
                if stranded
                    value ? (return rscaffold_bitarray = trues(scaffold_length,2)) : (return rscaffold_bitarray = falses(scaffold_length,2))
                else
```

```
                value ? (return rscaffold_bitarray = trues(scaffold_length,1)) : (return rscaffold_bitarray = falses(scaffold_length,1))
            end
        end

        function project_features_to_bitarray!(scaffold_feature_sf::SubDataFrame, scaffold_bitarray::BitArray)
            @inbounds for item in eachrow(scaffold_feature_sf)
                scaffold_bitarray[item.Start:item.End,1] = [true for base in item.Start:item.End]
                if size(scaffold_bitarray)[2] == 2 #if the bitarray is stranded
                    if item.Strand == '+'
                        scaffold_bitarray[item.Start:item.End,2] = [true for base in item.Start:item.End]
                    end
                end
            end
        end

        function subtract_features_from_bitarray!(scaffold_feature_sf::DataFrame, scaffold_bitarray::BitArray)
            @inbounds for item in eachrow(scaffold_feature_sf)
                scaffold_bitarray[item.Start:item.End,1] = [false for base in item.Start:item.End]
            end
        end

        function get_feature_df_from_bitarray(scaffold_id::String, scaffold_bitarray::BitArray)
            size(scaffold_bitarray)[2] == 2 ? scaffold_feature_df = DataFrame(SeqID = String[], Start = Integer[], End = Integer[], Strand=Char[]) : scaffold_feature_df = DataFrame(SeqID = String[], Start = Integer[], End = Integer[])

            new_feature_start = findnext(view(scaffold_bitarray,:,1), 1)
            while new_feature_start ≠ nothing # while new features are still found on the bitarray
```

```
                        if size(scaffold_bitarray)[2] == 2 #if stranded, get strand info
                            scaffold_bitarray[new_feature_start,2] == 1 ? new_feature_strand = '+' : new_feature_strand = '-'
                        end
                        if findnext(!eval,view(scaffold_bitarray,:,1),new_feature_st ≠ nothing
                            new_feature_end = findnext(!eval,view(scaffold_bitarray,:,1),new_featur #find next false after feature start and subtract 1 for feature end
                        else
                            new_feature_end = size(scaffold_bitarray)[1] #if none is found, the end of the feature is the end of hte scaffold
                        end
                        size(scaffold_bitarray)[2] == 2 ? push!(scaffold_feature_df,[scaffold_id, new_feature_start, new_feature_end, new_feature_strand]) : push!(scaffold_feature_df,[scaffold_id, new_feature_start, new_feature_end]) #push stranded feature info as appropriate
                        new_feature_start = findnext(view(scaffold_bitarray,:,1), new_feature_end+1)
                    end
                    return scaffold_feature_df
                end
```

15.2.12 /src/genome_sampling/sequence_sampler.jl

```
####SAMPLING FUNCTIONS####
#function for a Distributed worker to produce a set of samples of given parameters from genomic sequences
function get_sample_set(input_sample_jobs::RemoteChannel, completed_sample_jobs::RemoteChannel, progress_updates::RemoteChannel)
    while isready(input_sample_jobs)
```

```
        genome_path, genome_index_path, partition_df, partitionid, sample_set_length, sample_window_min, sample_window_max, deterministic = take!(input_sample_jobs)

        stranded::Bool = get_strand_dict()[partitionid]
        scaffold_sequence_record_dict::Dict{String,LongSequence} = build_scaffold_seq_dict(genome_path, genome_index_path)

        sample_df = DataFrame(SampleScaffold=String[],SampleStart=Integer[],SampleEnd=Integer
        metacoordinate_bitarray = trues(partition_df.MetaEnd[end])
        sample_set_counter = 0

        while sample_set_counter < sample_set_length #while we don't yet have enough sample sequence
            sample_scaffold::String, sample_Start::Integer, sample_End::Integer, sample_metaStart::Integer, sample_metaEnd::Integer, sample_sequence::LongSequence, strand::Char = get_sample(metacoordinate_bitarray, sample_window_min, sample_window_max,  partition_df, scaffold_sequence_record_dict; stranded=stranded, deterministic=deterministic)
            push!(sample_df,[sample_scaffold, sample_Start, sample_End, sample_sequence, strand]) #push the sample to the df
            sample_length = sample_End - sample_Start + 1
            sample_set_counter += sample_length #increase the counter by the length of the sampled sequence
            metacoordinate_bitarray[sample_metaStart:sample_metaEnd] = [false for base in 1:sample_length] #mark these residues as sampled
            put!(progress_updates, (partitionid, min(sample_set_counter,sample_set_length)))
        end
        put!(completed_sample_jobs,(partitionid, sample_df))
    end
end
                #get_sample_set() SUBFUNCTIONS
                #function defining whether partitions respect stranding upon fetching sequence (ie is the sequence fetched in the feature strand orientation, or are we agnostic about the strand we sample?)
                function get_strand_dict()
```

```
        return Dict("exon"⇒true,"periexonic"⇒true,"intergenic"⇒false)
    end
    #function to obtain a dict of scaffold sequences from a FASTA reader
    function build_scaffold_seq_dict(genome_fa, genome_index)
        genome_reader = open(FASTA.Reader, genome_fa, index=genome_index)
        seq_dict::Dict{String, LongSequence} = Dict{String,FASTA.Record}()
        @inbounds for record in genome_reader
            id = FASTA.identifier(record)
            seq_dict[rectify_identifier(id)]=FASTA.sequence(record)
        end
        close(genome_reader)
        return seq_dict
    end

    # function to convert scaffold ID from that observed by the masked .fna to the more legible one observed by the GRCz11 GFF3
    function rectify_identifier(scaffold_id::String)
        if length(scaffold_id) ≥ 4 && scaffold_id[1:4] == "CM00" #marks chromosome scaffold
            chr_code = scaffold_id[5:10]
            chr_no = "$(Int((parse(Float64,chr_code)) - 2884.2))"
            return chr_no
        else
            return scaffold_id
        end
    end

    #function to produce a single sample from a metacoordinate set and the feature df
    function get_sample(metacoordinate_bitarray::BitArray, sample_window_min::Integer, sample_window_max::Integer, partition_df::DataFrame, scaffold_seq_dict::Dict{String,LongSequence}; stranded::Bool=false, deterministic::Bool=false)
        proposal_acceptance = false
        sample_metaStart = 0
```

```
                    sample_metaEnd = 0
                    sample_Start = 0
                    sample_End = 0
                    sample_sequence = LongSequence{DNAAlphabet{2}}("")
                    sample_scaffold = ""
                    strand = nothing

                    while proposal_acceptance == false
                        available_indices = findall(metacoordinate_bitarray) #find all unsampled indices
                        window = "FAIL"
                        start_index,feature_metaStart,feature_metaEnd, feature_length = 0,0,0,0
                        strand = '0'
                        while window == "FAIL"
                            start_index = rand(available_indices) #randomly choose from the unsampled indices
                            feature_metaStart, feature_metaEnd, strand = get_feature_params_from_metacoord(start_index, partition_df, stranded)
                            feature_length = length(feature_metaStart:feature_metaEnd) #find the metaboundaries of the feature the index occurs in
                            if feature_length ≥ sample_window_min #don't bother finding windows on features smaller than min
                                window = determine_sample_window(feature_metaStart, feature_metaEnd, start_index, metacoordinate_bitarray, sample_window_min, sample_window_max) #get an appropriate sampling window around the selected index, given the feature boundaries and params
                            end
                        end

                        sample_scaffoldid, sample_scaffold_start, sample_scaffold_end = meta_to_feature_coord(window[1],window[2],partition_df)
```

```
79
80                          proposal_sequence =
                             ↪  fetch_sequence(sample_scaffoldid,
                             ↪  scaffold_seq_dict, sample_scaffold_start,
                             ↪  sample_scaffold_end, strand;
                             ↪  deterministic=deterministic) #get the
                             ↪  sequence associated with the sample window
81
82                          if mask_check(proposal_sequence)
83                              proposal_acceptance = true #if the sequence
                                 ↪  passes the mask check, accept the
                                 ↪  proposed sample
84                              sample_scaffold = sample_scaffoldid
85                              sample_Start = sample_scaffold_start
86                              sample_End = sample_scaffold_end
87                              sample_metaStart = window[1]
88                              sample_metaEnd = window[2]
89                              sample_sequence=proposal_sequence
90                          end
91                      end
92                      return sample_scaffold, sample_Start, sample_End,
                         ↪  sample_metaStart, sample_metaEnd,
                         ↪  sample_sequence, strand
93                  end
94
95                  #function to find a valid sampling window
96                  function
                     ↪  determine_sample_window(feature_metaStart::Integer,
                     ↪  feature_metaEnd::Integer, metacoord::Integer,
                     ↪  metacoord_bitarray::BitArray,
                     ↪  sample_window_min::Integer,
                     ↪  sample_window_max::Integer)
97                      window_start = 0
98                      window_end = 0
99                      feature_size = feature_metaEnd - feature_metaStart +
                         ↪  1
100                     feature_sampled = any(!eval,
                         ↪  metacoord_bitarray[feature_metaStart:feature_metaEnd])
101                     #attempt to construct the sample window by taking the
                         ↪  whole feature
102                     if !feature_sampled && feature_size <
                         ↪  sample_window_max && feature_size >
                         ↪  sample_window_min
```

```
                        window_start = feature_metaStart
                        window_end = feature_metaEnd
                        return window_start, window_end
                    else #if this fails, build the biggest window we can from the sampling point, within the feature
                        featurepos = metacoord - feature_metaStart + 1
                        next_sampled_index = findnext(!eval, metacoord_bitarray[feature_metaStart:feature_metaEnd], featurepos)
                        prev_sampled_index = findprev(!eval, metacoord_bitarray[feature_metaStart:feature_metaEnd], featurepos)
                        next_sampled_index === nothing ? (window_end = feature_metaEnd) : (window_end = next_sampled_index + feature_metaStart - 1)
                        prev_sampled_index === nothing ? (window_start = feature_metaStart) : (window_start = prev_sampled_index + feature_metaStart - 1)
                        windowsize = window_end - window_start + 1
                        #check to see if this window is bigger than the min, if not, return a failure code
                        windowsize < sample_window_min && return "FAIL"
                        #check to see if this window is bigger than the max, if so, trim it before returning, removing as evenly as possible around the metacoordinate
                        if windowsize > sample_window_max
                            bases_to_trim = windowsize - sample_window_max
                            clearance_5P = metacoord - window_start + 1
                            clearance_3P = window_end - metacoord + 1
                            trimmed=0
                            while trimmed < bases_to_trim
                                if clearance_5P ≥ clearance_3P && clearance_5P > 0
                                    clearance_5P -= 1
                                elseif clearance_3P > clearance_5P && clearance_3P > 0
                                    clearance_3P -= 1
                                end
                                trimmed +=1
                            end
                            window_start = metacoord - clearance_5P + 1
```

```
                            window_end = metacoord + clearance_3P - 1
                        end
                        return window_start, window_end
                    end
                end


                # function to check for repetitive stretches or degenerate bases in proposal sequence
                function mask_check(proposal_sequence::LongSequence)
                    proposal_acceptance = true
                    if hasambiguity(proposal_sequence) || isrepetitive(proposal_sequence, (length(proposal_sequence) ÷ 10))
                        proposal_acceptance = false
                    end
                    return proposal_acceptance
                end

####SHARED SEQUENCE FETCHER####
# function to get proposal sequence from dict of scaffold sequences, given coords and scaffold id
function fetch_sequence(scaffold_id::String, scaffold_seq_dict::Dict{String, LongSequence}, proposal_start::Integer, proposal_end::Integer, strand::Char; deterministic=false)
    if strand == '0' #unstranded samples may be returned with no preference in either orientation
        deterministic ? strand = '+' : (rand(1)[1] ≤ .5 ? strand = '+' : strand = '-')
    end
    if strand == '+'
        proposal_sequence = scaffold_seq_dict[scaffold_id][proposal_start:proposal_end]
    elseif strand == '-'
        proposal_sequence = reverse_complement(scaffold_seq_dict[scaffold_id][proposal_start:proposal
    else
        throw(ArgumentError("Invalid sample code! Must be '+', '-', or '0' (random strand)"))
    end

    return proposal_sequence
```

```
end

####SHARED BASIC COORDINATE SUBFUNCTIONS####
# function to push scaffold ID, start, and end points of given featuretype to supplied dataframe
function build_feature_df!(GFF3_path::String, feature_type::String, scaffold_exclusion::String, feature_df::DataFrame)
    reader = open(GFF3.Reader, GFF3_path) # access the GFF3
    @inbounds for record in reader # iterate over Gff3 records
        if GFF3.isfeature(record) # if the record is a feature
            if GFF3.featuretype(record) == feature_type #if the features is of the requested type, get the following info
                seqID = GFF3.seqid(record)
                if seqID ≠ scaffold_exclusion
                    seq_start = GFF3.seqstart(record)
                    seq_end = GFF3.seqend(record)
                    if feature_type == "CDS" || feature_type == "gene"
                        seq_strand = convert(Char, GFF3.strand(record))
                        push!(feature_df, [seqID, seq_start, seq_end, seq_strand])
                    else
                        push!(feature_df, [seqID, seq_start, seq_end]) # push relevant info to the df
                    end
                end
            end
        end
    end
    close(reader)
end

#function to assemble dataframe of scaffold coords + metacoords given gff3
function build_scaffold_df(gff3_path)
    scaffold_df = DataFrame(SeqID = String[], Start = Integer[], End = Integer[])
    build_feature_df!(gff3_path, "supercontig", "MT", scaffold_df)
    build_feature_df!(gff3_path, "chromosome", "MT", scaffold_df)
    add_metacoordinates!(scaffold_df)
    return scaffold_df
end

#function to add pad to either side of some featureset
```

```
function add_pad_to_coordinates!(feature_df::DataFrame, scaffold_df::DataFrame, pad_size::Integer; col_symbols::Array{Symbol}=[:Start, :End])
    pad_start_array = zeros(Integer,size(feature_df,1))
    pad_end_array = zeros(Integer,size(feature_df,1))
    feature_df[!, col_symbols[1]] = [max(feature_df.Start[i]-pad_size,1) for i in 1:size(feature_df,1)] #truncate pads at beginning and end of scaffolds
    feature_df[!, col_symbols[2]] = [min(feature_df.End[i]+pad_size,scaffold_df.End[findfirst(isequal(feature_df.
    for i in 1:size(feature_df,1)]
end

####SHARED METACOORDINATE FUNCTIONS####
# function to add a metacoordinate column to a dataframe of scaffold positions, allowing sampling across all scaffolds
function add_metacoordinates!(feature_df::DataFrame)
    meta_start_array = Integer[]
    meta_end_array = Integer[]
    metaposition = 1
    @inbounds for feature in eachrow(feature_df)
        push!(meta_start_array, (metaposition))
        metaposition += feature.End - feature.Start
        push!(meta_end_array, (metaposition))
        metaposition += 1
    end
    feature_df.MetaStart = meta_start_array # metacoordinate contains 'start' metaposition across all genomic material
    feature_df.MetaEnd = meta_end_array # metacoordinate contains 'end' metaposition across all genomic material
end

# function to convert metacoordinate set to scaffold-relative coordinates
function meta_to_feature_coord(meta_start::Integer, meta_end::Integer, feature_df::DataFrame)
    feature_row = get_feature_row_index(feature_df, meta_start)
    seqid = feature_df.SeqID[feature_row]
    scaffold_start = feature_df.Start[feature_row] + (meta_start - feature_df.MetaStart[feature_row])
    scaffold_end = feature_df.End[feature_row] - (feature_df.MetaEnd[feature_row] - meta_end)
    return seqid, scaffold_start, scaffold_end
end
```

```julia

#function to obtain the feature boundaries and strand of the feature that a metacoordinate falls within
function get_feature_params_from_metacoord(metacoordinate::Integer, feature_df::DataFrame, stranded::Bool)
    feature_row = get_feature_row_index(feature_df, metacoordinate)
    feature_metaStart = feature_df.MetaStart[feature_row]
    feature_metaEnd = feature_df.MetaEnd[feature_row]
    stranded ? feature_strand = feature_df.Strand[feature_row] : feature_strand = '0'
    return feature_metaStart, feature_metaEnd, feature_strand
end

#function obtain the index of a feature given its metacoordinate
function get_feature_row_index(feature_df::DataFrame, metacoordinate::Integer)
    total_feature_bases = feature_df.MetaEnd[end]
    if metacoordinate == total_feature_bases #edge case of metacoord at end of range
        return size(feature_df,1) #last index in df
    else
        index = findfirst(end_coord->end_coord>metacoordinate,feature_df.MetaEnd) #find the index of the feature whose end metacoord is > the query metacoord
        if index > 1 && metacoordinate == feature_df.MetaEnd[index-1] #if the metacoordinate is the last base of the previous feature
            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
            end
        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
        else
            throw(DomainError("Unexpected metacoordinate $metacoordinate in partition of $total_feature_bases bases, with feature start $(feature_df.MetaStart[index]), end $(feature_df.MetaEnd[index])"))
        end
    end
```

```
end
```

15.2.13 /src/likelihood_funcs/bg_lh_matrix.jl

```
#function to obtain positional likelihoods for a sequence under a given BHMM.
function get_BGHMM_symbol_lh(seq::AbstractMatrix, hmm::BHMM)
    @assert size(seq)[1] == 1
    (seq=Array(transpose(seq))) # one sequence at a time only
    symbol_lhs = zeros(length(seq))
    length_mask = [length(seq)-1]

    lls = bw_llhs(hmm, seq) #obtain log likelihoods for sequences and states
    log_α = messages_forwards_log(hmm.a, hmm.A, lls, length_mask) #get forward messages
    log_β = messages_backwards_log(hmm.A, lls, length_mask) # get backwards messages

    #calculate observation probability and γ weights
    K,Tmaxplus1,Strand = size(lls) #the last T value is the 0 end marker of the longest T

    #transforms to cut down log_ξ, log_γ assignment times
    lls = permutedims(lls, [2,1,3]) # from (K,T) to (T,K)
    log_α = permutedims(log_α, [2,1,3])
    log_β = permutedims(log_β, [2,1,3])

    log_γ = fill(-Inf, Tmaxplus1,K)
    log_pobs = logsumexp(lps.(log_α[1,:], log_β[1,:]))

    @inbounds for i = 1:K, t = 1:length_mask[1]
            log_γ[t,i] = lps(log_α[t,i],log_β[t,i],-log_pobs)
    end

    for t in 1:length_mask[1]
        symbol_lh::AbstractFloat = -Inf #ie log(p=0)
        for k = 1:K #iterate over states
                state_symbol_lh::AbstractFloat = lps(log_γ[t,k], log(hmm.B[k].p[seq[t]])) #state symbol likelihood is the γ weight * the state symbol probability (log implementation)
```

```
                symbol_lh = logaddexp(symbol_lh, state_symbol_lh) #sum the probabilities over states
        end
        symbol_lhs[t] = symbol_lh
    end

    return symbol_lhs[1:end-1] #remove trailing index position
end

#function to calculate BGHMM from an observation set and a dict of BGHMMs
function BGHMM_likelihood_calc(observations::DataFrame, BGHMM_dict::Dict, code_partition_dict = get_partition_code_dict(false); symbol=:PadSeq)
    lh_matrix_size = ((findmax(length.(collect(values(observations[!, symbol]))))[1]), length(observations[!, symbol]))
    BGHMM_lh_matrix = zeros(lh_matrix_size) #T, Strand, O

    BGHMM_fragments = fragment_observations_by_BGHMM(observations[!, symbol], observations.MaskMatrix)


    #@showprogress 1 "Writing frags to matrix.."
    Threads.@threads for (jobid, frag) in BGHMM_fragments
        (frag_start, o, partition, strand) = jobid

        partition_BGHMM::BHMM = BGHMM_dict[code_partition_dict[partition]]
        no_symbols = length(partition_BGHMM.B[1].p)
        order = Int(log(4,no_symbols) - 1)

        order_seq = get_order_n_seqs([frag], order)
        coded_seq = code_seqs(order_seq)

        subseq_symbol_lh = get_BGHMM_symbol_lh(coded_seq, partition_BGHMM)

        if strand == -1
            subseq_symbol_lh = reverse(subseq_symbol_lh)
        end #positive, unstranded frags  are inserted as-is
        BGHMM_lh_matrix[frag_start:frag_start+length(frag)-1,o] = subseq_symbol_lh
    end

    return BGHMM_lh_matrix
```

```
end

function fragment_observations_by_BGHMM(seqs::AbstractVector, masks::AbstractVector)
    likelihood_jobs = Vector{Tuple{Tuple,LongSequence{DNAAlphabet{2}}}}()
    @showprogress 1 "Fragmenting observations by partition..." for (o, obs_seq) in enumerate(seqs)
        mask = masks[o]
        frags = Vector{LongSequence{DNAAlphabet{2}}}() #container for all subsequences in observation
        frag = LongSequence{DNAAlphabet{2}}()

        frag_end=0
        frag_start = 1

        while frag_start < length(obs_seq) # while we're not at the sequence end
            curr_partition = mask[frag_start,1] #get the partition code of the frag start
            curr_strand = mask[frag_start,2] #get the strand of the frag start

            #JOBID COMPOSED HERE
            jobid = (frag_start, o, curr_partition, curr_strand) #compose an identifying index for this frag

            findnext(!isequal(curr_partition),mask[:,1],frag_start) ≠ nothing ? frag_end = findnext(!isequal(curr_partition),mask[:,1],frag_start) -1 : frag_end = length(obs_seq) #find the next position in the frag that has a different partition mask value from hte current one and set that position-1 to frag end, alternately frag end is end of the overall  sequence
            frag = obs_seq[frag_start:frag_end] #get the frag bases
            if curr_strand == -1 #if the fragment is reverse stranded
                reverse_complement!(frag) #use the reverse complement sequence
            end

            push!(likelihood_jobs,(jobid, frag)) #put the frag in the jobs vec
            frag_start = frag_end + 1 #move on
        end
```

```
    end
    return likelihood_jobs
end
```

15.2.14 /src/likelihood_funcs/hmm.jl

```
function obs_lh_given_hmm(observations, hmm; linear=true)
    linear ? (return linear_likelihood(observations, hmm)) : (return bw_likelihood(Matrix(transpose(observations)), hmm))
end

function bw_likelihood(observations, hmm)
    obs_lengths = [findfirst(iszero,observations[:,o])-1 for o in 1:size(observations,2)]

    lls = bw_llhs(hmm, observations)
    log_α = messages_forwards_log(hmm.a, hmm.A, lls, obs_lengths)
    log_β = messages_backwards_log(hmm.A, lls, obs_lengths)

    O = size(lls)[3] #the last T value is the 0 end marker of the longest T
    log_pobs = zeros(O)
    Threads.@threads for o in 1:O
        log_pobs[o] = logsumexp(lps.(log_α[:,1,o], log_β[:,1,o]))
    end

    return sum(log_pobs)
end

function linear_likelihood(observations,hmm)
    O= size(observations,1);
    obs_lengths = [findfirst(iszero,observations[o,:])-1 for o in 1:size(observations,1)] #mask calculations here rather than mle_step to prevent recalculation every iterate

    a = transpose(log.(hmm.a)); A = log.(hmm.A)
    N = length(hmm.B)
    mask=observations.≠0
    #INITIALIZATION
    βoi_T = zeros(O,N); βoi_t = zeros(O,N) #log betas at T initialised as zeros

```

```
    #RECURRENCE
    βoi_T = backwards_lh_sweep!(hmm, A, N, βoi_T, βoi_t, observations, mask, obs_lengths)

    #TERMINATION
    lls = c_llhs(hmm,observations[:,1])
    α1om = lls .+ a #first position forward msgs

    return lps([logsumexp(lps.(α1om[o,:], βoi_T[o,:])) for o in 1:O])
end
                #LINEAR_STEP SUBFUNCS
                function backwards_lh_sweep!(hmm, A, N, βoi_T, βoi_t, observations, mask, obs_lengths)
                    for t in maximum(obs_lengths)-1:-1:1
                        last_β=copy(βoi_T)
                        lls = c_llhs(hmm,observations[:,t+1])
                        omask = findall(mask[:,t+1])
                        βoi_T[omask,:] .+= view(lls,omask,:)
                        Threads.@threads for m in 1:N
                            βoi_t[omask,m] = logsumexp.(eachrow(view(βoi_T,omask,:).+transpose(vie
                        end
                        βoi_T=copy(βoi_t)
                    end
                    return βoi_T
                end
```

15.2.15 /src/reports/chain_report.jl

```
struct Chain_Report
    id::Chain_ID
    final_hmm::BHMM
    test_lh::AbstractFloat
    naive_lh::AbstractFloat
    final_delta::AbstractFloat
    state_run_lengths::AbstractVector{AbstractFloat}
    convergence_values::Chains
    convergence_diagnostic::ChainDataFrame
    converged::Bool
end

function Base.show(io::IO, report::Chain_Report)
```

```
    nominal_dict=Dict(0⇒"th",1⇒"st",2⇒"nd",3⇒"rd",4⇒"th",5⇒"th")
    haskey(nominal_dict,report.id.order) ? (nom_str=nominal_dict[report.id.order]) : (nom_str="th")
    printstyled(io, stdout, "BHMM EM Chain Results\n"; bold=true)
    println(io, "$(report.id.K)-state, $(report.id.order)$nom_str order BHMM")
    println(io, "Trained on observation set \"$(report.id.obs_id)\"")
    report.test_lh > report.naive_lh ? (lh_str="greater"; lh_color=:green) : (lh_str="less"; lh_color=:red)
    printstyled(io, "Test logP(O|θ): $(report.test_lh), $lh_str than the naive model's $(report.naive_lh)\n"; color=lh_color)
    println(io, "Replicate $(report.id.replicate)")
    report.converged ? println(io, "Converged with final step δ $(report.final_delta)") : println(io, "Failed to converge!")
    println(io, "────────────────────────────────────────────────────────────────────────────────────────────────────────────")
    println(io, "Last θ:")
    show(report.final_hmm)
    println(io, "State Mean Feature Length (bp)")
    for i in 1:length(report.state_run_lengths)
        println(io, "K$i: $(report.state_run_lengths[i].*float(report.id.order+1))")
    end
    println(io, "────────────────────────────────────────────────────────────────────────────────────────────────────────────")

    zidx=findfirst(!iszero,report.convergence_values["logP(O|θ)"].data)[1]

    lh_vec=Vector([report.convergence_values["logP(O|θ)"].data ... ][zidx:end])

    lh_plot=lineplot([zidx:length(lh_vec)+zidx-1 ... ],lh_vec;title="Chain likelihood evolution", xlabel="Training iterate", xlim=(0,length(lh_vec)+zidx+1), ylim=(floor(minimum(lh_vec),sigdigits=2),ceil(maximum(lh_vec),sigdigits=2)), name="logP(O|θ)")
    lineplot!(lh_plot,[report.naive_lh for i in 1:length(lh_vec)], color=:magenta,name="naive")
    show(io, lh_plot)
    println(io, "\n")

    k1vec=Vector([report.convergence_values["K1"].data ... ])
```

```
    k_plot=lineplot(k1vec, title="State p(Auto) evolution", xlabel="Training iterate", ylabel="prob", name="K1",ylim=(0,1),xlim=(0,length(k1vec)))
    for k in 2:report.id.K
        kvec=Vector([report.convergence_values["K$k"].data ... ])
        lineplot!(k_plot,kvec,name="K$k")
    end
    show(io,k_plot)
    println(io)

    printstyled(io,"\nConvergence Diagnostics\n",bold=true)
    if report.convergence_diagnostic.name == "short"
        printstyled(io,"Convergence diagnostics unavailable for chains <10 steps!\n", color=:yellow)
    elseif report.convergence_diagnostic.name == "error"
        printstyled(io,"Convergence diagnostics errored! Zeros in autotransition matrix?\n", color=:red)
    else
        all(Bool.(report.convergence_diagnostic.nt.stationarity)) && all(Bool.(report.convergence_diagnostic.nt.test)) ? printstyled(io, "Likelihood and autotransition probabilites converged and passing tests.\n", color=:green) : printstyled(io, "Not all parameters converged or passing tests!\n",color=:red)
        display(report.convergence_diagnostic)
    end
end

function latex_report(report::Chain_Report)

end

function report_chains(chains::Dict{Chain_ID,Vector{EM_step}}, test_sets::Dict{String,Vector{LongSequence{DNAAlphabet{2}}}}; naive_hmm=BHMM(ones(1,1),[Categorical(4)]))
    job_ids=[id for (id,chain) in chains] # make list of chain_ids
    partitions=unique([id.obs_id for id in job_ids]) #get list of unique partitions represented among chain_ids
    naive_order = Integer(log(4,length(naive_hmm.B[1].support))-1)
    for partition in partitions #check that all partitions are available for testing and make sure codes for naive hmm are generated by adding job_ids
```

```
        !(partition in keys(test_sets)) && throw(ArgumentError("Observation set $partition required by chains for testing not present in test sets!"))
        push!(job_ids, Chain_ID(partition,length(naive_hmm.a),naive_order, 1))
    end

    code_dict=code_job_obs(job_ids, test_sets) #code all the obs sets that are necessary ot perform our tests
    naive_lhs=Dict{String,Float64}() #construct dict of naive likelihoods for all partitions to be tested prior to individual tests
    @showprogress 1 "Testing naive hmm..." for ((partition,order),obs_set) in code_dict
        order==naive_order && (naive_lhs[partition]=obs_lh_given_hmm(code_dict[(partition,naive_order)]
    end

    reports=Dict{Chain_ID,Chain_Report}()
    @showprogress 1 "Testing chains..." for (id, chain) in chains
        chains_array=zeros(length(chain),1+id.K)
        for (n,step) in enumerate(chain)
            chains_array[n,1]=step.log_p
            chains_array[n,2:end]=get_diagonal_array(step.hmm)
        end
        convergence_values=Chains(chains_array,["logP(O|θ)", ["K"*string(i) for i in 1:id.K]...])
        if length(chain)≥10
            try
                convergence_diagnostic=heideldiag(convergence_values)[1]
            catch convergence_diagnostic=ChainDataFrame("error",(;:a⇒[1.]))
            end
        else
            convergence_diagnostic=ChainDataFrame("short",(;:a⇒[1.]))
        end

        id.K > 1 ? (mean_run_lengths=sim_run_lengths(get_diagonal_array(chain[end].hmm),1000 : (mean_run_lengths=[Inf])
        test_lh=obs_lh_given_hmm(code_dict[(id.obs_id,id.order)], chain[end].hmm)
```

```
        reports[id]=Chain_Report(id, chain[end].hmm, test_lh, naive_lhs[id.obs_id], chain[end].delta, mean_run_lengths, convergence_values, convergence_diagnostic, chain[end].converged)
    end

    return reports
end

function get_diagonal_array(hmm::BHMM)
    k = length(hmm.a)
    diagonal = zeros(k)
    @inbounds for i in 1:k
        diagonal[i] = hmm.A[i,i]
    end
    return diagonal
end

#function to simulate run lengths for vector of diagonal values
function sim_run_lengths(diagonal_value::AbstractArray, samples::Integer)
    mean_run_lengths = zeros(length(diagonal_value))
    for (i, value) in enumerate(diagonal_value)
        if value == 0.
            mean_run_lengths[i]=0.
        elseif value ==1.
            mean_run_lengths[i]=Inf
        else
            runlengths = zeros(Integer, samples)
            for s in 1:samples
                run = true
                runlength = 0
                while run
                    runlength += 1
                    if rand(1)[1] > value
                        run = false
                    end
                end
                runlengths[s] = runlength
            end
            mean_run_lengths[i] = mean(runlengths)
        end
    end
    return mean_run_lengths
```

```
end
```

15.2.16 /src/reports/partition_report.jl

```
struct Order_Report
    converged_K::Vector{Float64}
    converged_lh::Vector{Float64}
    failed_K::Vector{Float64}
    failed_lh::Vector{Float64}
end

struct Partition_Report
    partition_id::String
    naive_lh::Float64
    orddict::Dict{Integer,Order_Report}
    best_model::Tuple{Chain_ID,BHMM}
    best_repset::Vector{Chain_ID}
end

function Base.show(io::IO, report::Partition_Report)
    printstyled(io, "BACKGROUND HMM TRAINING REPORT\n", bold=true)
    printstyled(io, "Genome partition id: $(report.partition_id)\n",
    ↪ bold=true, color=:magenta)

    for (order, ordreport) in report.orddict
        if length(ordreport.failed_lh)>0
            ordplot=scatterplot(ordreport.converged_K,
            ↪ ordreport.converged_lh; name="Converged", title="Order
            ↪ $order HMMs", xlabel="K# States", ylabel="P(O|θ)",
            ↪ color=:green, xlim=(1,maximum(ordreport.converged_K)),
            ↪ ylim=(floor(min(minimum(ordreport.converged_lh),minimum(ordreport.fai
            length(ordreport.failed_K)≥1 && scatterplot!(ordplot,
            ↪ ordreport.failed_K, ordreport.failed_lh;
            ↪ name="Unconverged",color=:red)
            lineplot!(ordplot, [report.naive_lh for i in
            ↪ 1:maximum(ordreport.converged_K)],
            ↪ name="Naive",color=:magenta)
            show(ordplot)
            println()
        else
```

```
            ordplot=scatterplot(ordreport.converged_K, ordreport.converged_lh; name="Converged", title="Order $order HMMs", xlabel="K# States", ylabel="P(O|θ)", color=:green, xlim=(1,maximum(ordreport.converged_K)), ylim=(floor(min(minimum(ordreport.converged_lh),report.naive_lh),sigd
            lineplot!(ordplot, [report.naive_lh for i in 1:maximum(ordreport.converged_K)], name="Naive",color=:magenta)
            show(ordplot)
            println()
        end
    end
end

function report_partitions(chain_reports::Dict{Chain_ID,Chain_Report})
    partitions=Vector{String}()
    orders=Vector{Integer}()
    Ks=Vector{Integer}()
    reps=Vector{Integer}()
    for id in keys(chain_reports)
        !in(id.obs_id, partitions) && push!(partitions, id.obs_id)
        !in(id.order, orders) && push!(orders, id.order)
        !in(id.K, Ks) && push!(Ks, id.K)
        !in(id.replicate, reps) && push!(reps, id.replicate)
    end

    reports=Vector{Partition_Report}()
    for partition in partitions
        best_lh,best_rep=-Inf,Chain_ID("empty",1,0,1)
        naive_lh=1.
        orddict=Dict{Integer,Order_Report}()
        for order in orders
            conv_statevec=Vector{Float64}()
            fail_statevec=Vector{Float64}()
            conv_lh_vec=Vector{Float64}()
            fail_lh_vec=Vector{Float64}()
            for (id,report) in chain_reports
                if id.obs_id==partition && id.order==order
                    report.test_lh > best_lh && (best_lh=report.test_lh;best_rep=id)
                    naive_lh==1. && (naive_lh=chain_reports[id].naive_lh)
                    chain_reports[id].converged ? (push!(conv_statevec,id.K);
```

```
                    push!(conv_lh_vec,chain_reports[id].test_lh)) : (push!(fail_statevec,id.K);
                    push!(fail_lh_vec,chain_reports[id].test_lh))
                end
            end

            orddict[order]=Order_Report(conv_statevec,conv_lh_vec,fail_statevec,f
        end
        best_model=(best_rep,chain_reports[best_rep].final_hmm)
        best_repset=[Chain_ID(partition, best_rep.K, best_rep.order, rep) for rep in reps]

        push!(reports,Partition_Report(partition,naive_lh,orddict,best_model,best_
    end
    return reports
end
```

15.2.17 /src/reports/replicate_convergence.jl

```
struct Replicate_Report
    ids::Vector{Chain_ID} #replicates
    state_vecs::Dict{Chain_ID,Matrix{Float64}} #vectors of autotransition probabilities evolving by iterate, concatanated to matrices, indexed by Chain_ID
    emission_arrays::Dict{Chain_ID,Array{Float64}} #iterate*symbol*state_vecs
    sorted_id1_states::Vector{Integer}
    sort_dicts::Dict{Chain_ID,Dict{Integer,Integer}}#emission channels sorted on the basis of euclidean closeness to ids[1]
    sorted_symbols::Dict{Integer,Vector{Integer}}


    Replicate_Report(ids,state_vecs,emission_arrays, sorted_id1_states,sort_dicts,sorted_symbols) = assert_repreport(ids) && new(ids,state_vecs,emission_arrays,sorted_id1_states,sort_dicts,sorted_symbol
end

function assert_repreport(ids::Vector{Chain_ID})
    Ks=Vector{Int64}()
    orders=Vector{Int64}()
    replicates=Vector{Int64}()
```

```
    obsids=Vector{String}()

    for id in ids
        push!(Ks,id.K)
        push!(orders,id.order)
        push!(obsids,id.obs_id)
        push!(replicates, id.replicate)
    end

    length(unique(Ks)) > 1 && throw(ArgumentError("Replicate set includes chains with different K state numbers! All chains must have HMMs with the same number of states."))
    length(unique(orders)) > 1 && throw(ArgumentError("Replicate set includes chains with different order coding numbers! All chains must have HMMs with the same order coding."))
    !allunique(replicates) && throw(ArgumentError("Replicate set includes chains with the same replicate #. Replicates should be unique!"))

    return true
end

function report_replicates(repset::Vector{Chain_ID},chains::Dict{Chain_ID,Vector{EM_step}}
    assert_repreport(repset) #check that the repset will pass its constructor before doing the work
    emission_arrays=Dict{Chain_ID,AbstractArray{AbstractFloat}}()
    state_arrays=Dict{Chain_ID,AbstractArray{AbstractFloat}}()

    for id in repset
        emission_array=zeros(length(chains[id]),4^(id.order+1),id.K)
        state_array=zeros(length(chains[id]),id.K)
        for (it, step) in enumerate(chains[id])
            for k in 1:id.K
                emission_array[it,:,k]=step.hmm.B[k].p
                state_array[it,k]=step.hmm.A[k,k]
            end
        end
        emission_arrays[id]=emission_array
        state_arrays[id]=state_array
    end

    println(repset)
    println(emission_arrays)
```

```
    println(state_arrays)

    sort_dicts,sorted_id1_states,sorted_symbols=sort_emitters_by_distance!(repset

    return Replicate_Report(repset,state_arrays,emission_arrays,sorted_id1_states,sort_dicts,sorted_symbols)
end

function sort_emitters_by_distance!(repset,emission_arrays)
    id1=repset[1]
    length(repset)==1 && (return Dict{Chain_ID,Dict{Integer,Integer}}(),[1:id1.K ... ],[1:4^id1.order+1])

    final_emissions = zeros(length(repset),size(emission_arrays[id1])[2:3] ... )
    for (n,id) in enumerate(repset)
        final_emissions[n,:,:] = emission_arrays[id][end,:,:]
    end

    distances=zeros(length(repset)-1,id1.K,id1.K)
    for rep in 1:length(repset)-1
        for k in 1:id1.K, j in 1:id1.K

            distances[rep,k,j]=euclidean(final_emissions[1,:,k],final_emissions[1
        end
    end

    sortdicts=Dict{Chain_ID,Dict{Integer,Integer}}()
    sorted_id1_states=Vector{Integer}()
    sorted_symbols=Dict{Integer,Vector{Integer}}()
    for rep in 1:length(repset)-1
        sortdicts[repset[rep+1]]=Dict{Integer,Integer}()
    end

    for i in 1:id1.K
        id1_mindist_K=findmin([sum([minimum(distances[rep,k,:]) for rep in 1:length(repset)-1]) for k in 1:id1.K])[2]#find the state from the first replicate that has minimum cumulative distance to the closest state in the other replicates, excluding already-chosen states by spiking their values
        push!(sorted_id1_states,id1_mindist_K)
        for rep in 1:length(repset)-1
```

```
            rep_k=findmin(distances[rep,id1_mindist_K,:])[2]
            sortdicts[repset[rep+1]][id1_mindist_K]=rep_k
            distances[rep,:,rep_k]•=1.0 #mask
            distances[1,id1_mindist_K,:]•=1.0 #mask
        end


        symbol_distances=sum(hcat([euclidean.(final_emissions[1,:,id1_mindist_K],
        for (n,id) in enumerate(repset[2:end])] ... ),dims=2)
        sorted_symbols[id1_mindist_K]=sortperm(symbol_distances[:,1])
    end

    return sortdicts,sorted_id1_states,sorted_symbols
end


function Base.show(io::IO, report::Replicate_Report;
    top_states::Integer=report.ids[1].K, top_symbols::Integer=2)
    printstyled(io, "REPLICATE CONVERGENCE REPORT\n", color=:green)
    println("────────────────────────────────────────────────────────────────────────────────")
    for i in 1:top_states
        id=report.ids[1]
        id1_state=report.sorted_id1_states[i]
        itmax=maximum([size(report.state_vecs[id],1) for id in
        report.ids])

        autoplt=lineplot(report.state_vecs[id][:,id1_state],
                        title="Autotransition prob. convergence",
                        name="$(report.ids[1]) K$id1_state",
                        xlim=(0,ceil(itmax, sigdigits=2)),
                        ylim=(0,1),
                        xlabel="iterate",
                        ylabel="p")


        symplot=lineplot(report.emission_arrays[id][:,report.sorted_symbols[id1_s

                        report.emission_arrays[id][:,report.sorted_symbols[id1_st
                        title="Symbol prob. convergence",
                        name="$(report.ids[1]) K$id1_state",
                        xlim=(0,1),
```

```
                        ylim=(0,1),
                        xlabel="Symbol 1 p",
                        ylabel="S2 p")

        for (n,id) in enumerate(report.ids)
            if n > 1
                lineplot!(autoplt,
                 ↪  report.state_vecs[id][:,report.sort_dicts[id][id1_state]],
                            name="$(report.ids[n])
                             ↪  K$(report.sort_dicts[id][id1_state])")
                lineplot!(symplot,

                             ↪  report.emission_arrays[id][:,report.sorted_symbols[id

                             ↪  report.emission_arrays[id][:,report.sorted_symbols[id
                            name="$(report.ids[n])
                             ↪  K$(report.sort_dicts[id][id1_state])")
            end
        end

        show(autoplt)
        println("\n")
        show(symplot)
        println("\n")
    end
end
```

15.2.18 /src/utilities/BBG_analysis.jl

```
#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)

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)]

```

```
    for step in chains[1]
        hmm=step[2]
        length(hmm.B)<3 && throw(ArgumentError, "3- or greater state hmms required")
        push!(coords[1], (hmm.B[1].p[1],hmm.B[2].p[1],hmm.B[3].p[1]))
    end

    for (c, chain) in enumerate(chains[2:end])
        ref_idxs=Vector{Integer}()
        ref_vecs=[chains[1][end][2].B[1].p,chains[1][end][2].B[2].p,chains[1][end
        end_hmm=chain[end][2]
        length(end_hmm.B)<3 && throw(ArgumentError, "3- or greater state hmms required")

        for vec in ref_vecs
            euclideans=Vector{AbstractFloat}()
            for d in 1:length(end_hmm.B)
                push!(euclideans, euclidean(vec, end_hmm.B[d].p))
            end
            min_euc_idx = findmin(euclideans)[2]
            while min_euc_idx in ref_idxs
                euclideans[min_euc_idx]=1.
                min_euc_idx = findmin(euclideans)[2]
            end
            push!(ref_idxs,findmin(euclideans)[2])
        end

        for step in chain
            hmm=step[2]
            push!(coords[c+1], (hmm.B[ref_idxs[1]].p[1],hmm.B[ref_idxs[2]].p[1],hmm.B[ref_idxs[3]].p
        end
    end

    return coords
end
```

15.2.19 /src/utilities/BBG_progressmeter.jl

```
#UTILITY PROGRESSMETER, REPORTS WORKER NUMBER AND CURRENT ITERATE
mutable struct ProgressHMM{T<:Real} <: ProgressMeter.AbstractProgress
```

```
    thresh::T
    dt::AbstractFloat
    val::T
    counter::Integer
    triggered::Bool
    tfirst::AbstractFloat
    tlast::AbstractFloat
    printed::Bool        # true if we have issued at least one status update
    desc::AbstractString # prefix to the percentage, e.g.  "Computing..."
    color::Symbol        # default to green
    output::IO           # output stream into which the progress is written
    numprintedvalues::Integer   # num values printed below progress in last iteration
    offset::Integer             # position offset of progress bar (default is 0)
    steptime::AbstractFloat
    function ProgressHMM{T}(thresh;
                                dt::Real=0.1,
                                desc::AbstractString="Progress: ",
                                color::Symbol=:green,
                                output::IO=stderr,
                                offset::Integer=0,
                                start_it::Integer=1) where T
        tfirst = tlast = time()
        printed = false
        new{T}(thresh, dt, typemax(T), start_it, false, tfirst, tlast, printed, desc, color, output, 0, offset, 0.0)
    end
end

ProgressHMM(thresh::Real, dt::Real=0.1, desc::AbstractString="Progress: ",
        color::Symbol=:green, output::IO=stderr;
        offset::Integer=0, start_it::Integer=1) = ProgressHMM{typeof(thresh)}(thresh, dt=dt, desc=desc, color=color, output=output, offset=offset, start_it=start_it)

ProgressHMM(thresh::Real, desc::AbstractString, offset::Integer=0, start_it::Integer=1) = ProgressHMM{typeof(thresh)}(thresh, desc=desc, offset=offset, start_it=start_it)
```

```julia

function update!(p::ProgressHMM, val, steptime; options...)
    p.val = val
    p.counter += 1
    p.steptime = steptime
    updateProgress!(p; options...)
end

function updateProgress!(p::ProgressHMM; showvalues = Any[], valuecolor = :blue, offset::Integer = p.offset, keep = (offset == 0))
    p.offset = offset
    t = time()
    if p.val ≤ p.thresh && !p.triggered
        p.triggered = true
        if p.printed
            p.triggered = true
            dur = ProgressMeter.durationstring(t-p.tfirst)
            msg = @sprintf "%s Time: %s (%d iterations)" p.desc dur p.counter
            print(p.output, "\n" ^ (p.offset + p.numprintedvalues))
            ProgressMeter.move_cursor_up_while_clearing_lines(p.output, p.numprintedvalues)
            ProgressMeter.printover(p.output, msg, p.color)
            ProgressMeter.printvalues!(p, showvalues; color = valuecolor)
            if keep
                println(p.output)
            else
                print(p.output, "\r\u1b[A" ^ (p.offset + p.numprintedvalues))
            end
        end
        return
    end

    if t > p.tlast+p.dt && !p.triggered
        elapsed_time = t - p.tfirst
        msg = @sprintf "%s (convergence: %g ⇒ %g, iterate: %g, step time: %s)" p.desc p.val p.thresh p.counter hmss(p.steptime)
        print(p.output, "\n" ^ (p.offset + p.numprintedvalues))
        ProgressMeter.move_cursor_up_while_clearing_lines(p.output, p.numprintedvalues)
        ProgressMeter.printover(p.output, msg, p.color)
        ProgressMeter.printvalues!(p, showvalues; color = valuecolor)
```

```
        print(p.output, "\r\u1b[A" ^ (p.offset + p.numprintedvalues))
        # Compensate for any overhead of printing. This can be
        # especially important if you're running over a slow network
        # connection.
        p.tlast = t + 2*(time()-t)
        p.printed = true
    end
end

            function hmss(dt)
                isnan(dt) && return "NaN"
                (h,r) = divrem(dt,60*60)
                (m,r) = divrem(r, 60)
                (isnan(h)||isnan(m)||isnan(r)) && return "NaN"
                string(Int(h),":",Int(m),":",Int(ceil(r)))
            end
```

15.2.20 /src/utilities/HMM_init.jl

```
function autotransition_init(K::Integer, order::Integer,
↪   partition::String="")
        a = rand(Dirichlet(ones(K)/K)) #uninformative prior on initial
        ↪   state probabilities
        A = strong_autotrans_matrix(K)
        no_emission_symbols = Int(4^(order+1)) #alphabet size for the
        ↪   order
        emission_dists = [generate_emission_dist(no_emission_symbols) for
        ↪   i in 1:K]
        #generate the BHMM with the appropriate transition matrix and
        ↪   emissions distributions
        hmm = BHMM(a, A, emission_dists, partition)
end
            #function to construct BHMM transition matrix with strong
            ↪   priors on auto-transition
            function strong_autotrans_matrix(states::Integer,
            ↪   prior_dope::AbstractFloat=(states*250.0),
            ↪   prior_background::AbstractFloat=.1)
                transition_matrix=zeros(states,states)
                for k in 1:states
                    dirichlet_params = fill(prior_background, states)
                    dirichlet_params[k] = prior_dope
```

```
                    transition_matrix[k,:] = rand(Dirichlet(dirichlet_params))
                end
                return transition_matrix
            end

            #function to construct BHMM state emission distribution from uninformative dirichlet over the alphabet size
            function generate_emission_dist(no_emission_symbols, prior=Dirichlet(ones(no_emission_symbols)/no_emission_symbols))
                return Categorical(rand(prior))
            end
```

15.2.21 /src/utilities/load_balancer.jl

```
struct LoadConfig
    k_range::UnitRange
    o_range::UnitRange
    blacklist::Vector{Chain_ID}
    whitelist::Vector{Chain_ID}
    function LoadConfig(k_range, o_range; blacklist=Vector{Chain_ID}(), whitelist=Vector{Chain_ID}())
        assert_loadconfig(k_range, o_range) && new(k_range, o_range, blacklist, whitelist)
    end
end

function assert_loadconfig(k_range,o_range)
    k_range[1] < 1 && throw(ArgumentError("Minimum value of LoadConfig k_range is 1!"))
    o_range[1] < 0 && throw(ArgumentError("Minimum value of LoadConfig o_range is 0!"))
    return true
end

#subfunc to handle balancing memory load on dissimilar machines in cluster
function load_balancer(no_models::Integer, hmm_jobs::RemoteChannel, config::LoadConfig, timeout::Integer=300)
    jobid::Chain_ID, start_iterate::Integer, hmm::BHMM, job_norm::AbstractFloat, observations::Matrix = take!(hmm_jobs)

```

```
    starttime=time()
    while (jobid.K < config.k_range[1] || jobid.K > config.k_range[end] || jobid.order < config.o_range[1] || jobid.order > config.o_range[end] || jobid in config.blacklist || (length(config.whitelist)>0 && !(jobid in config.whitelist))) && (time()-starttime ≤ timeout) #while a job prohibited by load table, keep putting the job back and drawing a new one
        put!(hmm_jobs, (jobid, start_iterate, hmm, job_norm, observations))
        jobid, start_iterate, hmm, job_norm, observations = take!(hmm_jobs)
    end
    (time()-starttime > timeout) ? (return 0,0,0,0,0) : (return jobid, start_iterate, hmm, job_norm, observations)
end
```

15.2.22 /src/utilities/log_prob_sum.jl

```
                #subfuncs to handle sums of log probabilities that may include -Inf (ie p=0), returning -Inf in this case rather than NaNs
                function lps(adjuvants)
                    prob = sum(adjuvants) ; isnan(prob) ? - Inf : prob
                end

                function lps(base, adjuvants... )
                    prob = base+sum(adjuvants) ; isnan(prob) ? -Inf : prob
                end
```

15.2.23 /src/utilities/model_display.jl

```
cs_dict=Dict('A'⇒:green,'C'⇒:blue, 'G'⇒:yellow, 'T'⇒:red)

function print_emitters(state::Categorical) #print informative symbols from the state's emission distribution, if any
    order=Integer(log(4,length(state.p))-1)
    alphabet=CompoundAlphabet(ACGT,order)
    bits=log(2,length(state.p)) #higher order hmms express more information per symbol
```

```
    dummy_p=state.p.+10^-99 #prevent -Inf or NaN from zero probability symbols
    infoscore=(bits+sum(x*log(2,x) for x in dummy_p))
    infovec=[x*infoscore for x in dummy_p]
    infovec./=bits
    print("<<< ")
    for (symbol,symbol_prob) in enumerate(infovec)
        if symbol_prob ≥ .05
            str=string(alphabet.integers[symbol])
            if symbol_prob ≥ .7
                for char in str
                    printstyled(stdout, char; bold=true, color=cs_dict[char])
                end
            elseif .7 > symbol_prob ≥ .25
                for char in str
                    printstyled(stdout, char; color=cs_dict[char])
                end
            elseif .25 > symbol_prob
                for char in str
                    printstyled(stdout, lowercase(char); color=cs_dict[char])
                end
            end
            print(" ")
        end
    end
    println(">>>")
end
```

15.2.24 /src/utilities/observation_coding.jl

```
#KMER ORDER/SEQUENCE INTEGER CODING UTILITIES
#higher order DNA alphabet
struct CompoundAlphabet
  symbols::Dict{Mer{DNAAlphabet{2}},Integer}
  integers::Dict{Integer,Mer{DNAAlphabet{2}}}
  #build a CompoundAlphabet for DNA of some order_no
    function CompoundAlphabet(alphabet::Tuple, order_no::Integer)
        symbols = Dict{Mer{DNAAlphabet{2}},Integer}()
        integers = Dict{Integer,Mer{DNAAlphabet{2}}}()

```

```
        tuples = Array{Tuple}
        if order_no > 0
            tuples = collect(Iterators.product([alphabet for order in 0:order_no]...))
        else #0th order compound product of an alphabet is that alphabet
            tuples = collect(Iterators.product(alphabet))
        end

        @inbounds for index in eachindex(tuples)
            tuple_seq = Mer{DNAAlphabet{2}}(collect(tuples[index]))
            integers[index] = tuple_seq
            symbols[tuple_seq] = index
        end

        new(symbols,integers)
    end
end


struct N_Order_ntSequence
    alphabet::CompoundAlphabet
    seq_lengths::Vector{Integer}
    order_kmers::Vector{Vector{Mer{DNAAlphabet{2}}}}
  end


function code_job_obs(job_ids::Vector{Chain_ID}, obs_sets::Dict{String,Vector{LongSequence{DNAAlphabet{2}}}})
    code_jobs=Vector{Tuple{String,Integer}}()
    for id in job_ids #assemble a vector of observation encoding jobs
        code_job=(id.obs_id, id.order)
        !(code_job in code_jobs) && push!(code_jobs, code_job)
    end

    code_dict = Dict{Tuple{String,Integer}, AbstractArray}()
    @showprogress 1 "Encoding observations..." for (obs_id, order) in code_jobs #build the appropriate sample sets once
        order_seqs = get_order_n_seqs(obs_sets[obs_id],order) #get the kmer sequences at the appropriate order
        coded_seqs = code_seqs(order_seqs, sorted=true) #numerically code the sequences in trainable format
        code_dict[(obs_id, order)] = coded_seqs
    end
```

```
    return code_dict
end

#from a vector of LongSequences, get
function get_order_n_seqs(seqs::Vector{LongSequence{DNAAlphabet{2}}},
↪   order_no::Integer, base_tuple::Tuple=ACGT)
    kmer_vecs = Vector{Vector{Mer{DNAAlphabet{2}}}}()
    length_vec = Vector{Integer}()
    window = order_no + 1

    for seq in seqs
        kmer_vec = Vector{Mer{DNAAlphabet{2}}}()
        @inbounds for (i, kmer) in
         ↪  collect(each(Mer{DNAAlphabet{2},window},seq))
            push!(kmer_vec, kmer)
        end

        push!(kmer_vecs, kmer_vec)
        push!(length_vec, length(seq))
    end

    return nordseqs = N_Order_ntSequence(CompoundAlphabet(base_tuple,
     ↪  order_no), length_vec, kmer_vecs)
end

#convert tuple kmers to symbol codes
function code_seqs(input::N_Order_ntSequence, offsets::Vector=[0 for i in
↪   1:length(input.order_kmers)]; sorted::Bool=false)
    symbol_no=length(input.alphabet.symbols)
    if symbol_no ≤ typemax(UInt8)
        integer_type = UInt8
    elseif typemax(UInt8) < symbol_no < typemax(UInt16)
        integer_type = UInt16
    else
        integer_type = UInt32
    end

    alphabet = input.alphabet
    output = zeros(integer_type,  length(input.order_kmers),
     ↪  (maximum([length(seq) for seq in input.order_kmers])+1)) #leave 1
     ↪  missing value after the longest sequence forindexing sequence
     ↪  length in CLHMM messages
    sorted && (sort_idxs = sortperm(input.seq_lengths,rev=true))
```

```
    sorted ? input_seqs = input.order_kmers[sort_idxs] : input_seqs = input.order_kmers

    for (i, seq) in enumerate(input_seqs)
        for t in 1:length(seq)
            curr_kmer = seq[t]
            curr_code = alphabet.symbols[curr_kmer]
            output[i,t+offsets[i]]=curr_code
        end
    end
    return output
end
```

15.2.25 /src/utilities/utilities.jl

```
"""
Utility functions for learning and using background genomic hidden markov models
"""
#function to split random sample dataframe into training and test sets (divide total sequence length by half)
function split_obs_sets(sample_dfs::Dict{String,DataFrame})
    training_sets = Dict{String,Vector{LongSequence{DNAAlphabet{2}}}}()
    test_sets = Dict{String,Vector{LongSequence{DNAAlphabet{2}}}}()

    for (partition_id, partition) in sample_dfs
        partition = partition[shuffle(1:size(partition, 1)),:] #shuffle the samples to avoid any effect from sampling without replacement
        partition.sampleLength = (partition.SampleEnd - partition.SampleStart) .+ 1
        midway = sum(partition.sampleLength)÷2
        split_index = 0
        counter = 0
        while split_index == 0
            counter += 1
            length_sum = sum(partition.sampleLength[1:counter])
            if length_sum > midway
                split_index = counter
            end
        end

```

```
        training_sets[partition_id]  = partition.SampleSequence[1:split_index-1]
        test_sets[partition_id] = partition.SampleSequence[split_index:end]
    end
    return training_sets, test_sets
end
```

15.2.26 /test/ref_fns.jl

```
#slow algo written in mimicry of Churbanov & Winters
function old_linear(hmm, observations, obs_lengths)
    O = size(observations)[2]
    a = log.(hmm.A); a = log.(hmm.a)
    N = length(hmm.B); B = length(hmm.B[1].support); b = [log(hmm.B[m].p[γ]) for m in 1:N, γ in 1:B]
    α1oi = zeros(O,N); β1oi = zeros(O,N); Eoi = zeros(O,N,B); Toij = zeros(O,N,N); Aoi = zeros(O,N); log_pobs=zeros(O); γt=0

    for o in 1:O
        #INITIALIZATION
        T = obs_lengths[o]; βoi_T = zeros(N) #log betas at T initialised as zeros
        EiT = fill(-Inf,B,N,N); TijT = fill(-Inf,N,N,N) #Ti,j(T,m) = 0 for all m; in logspace

        @inbounds for m in 1:N, i in 1:N, γ in 1:B
            observations[T, o] == γ && m == i ? (EiT[γ, i, m] = 0) :
                (EiT[γ, i, m] = -Inf) #log Ei initialisation
        end

        #RECURRENCE
        @inbounds for t in T-1:-1:1
            βoi_t = similar(βoi_T); Tijt = similar(TijT); Eit = similar(EiT)
            Γ = observations[t+1,o]; γt = observations[t,o]
            Γ==0 && println("hooo!! $o $t")
            for m in 1:N
                βoi_t[m] = logsumexp([lps(a[m,j], b[j,Γ], βoi_T[j]) for j in 1:N])
                for i in 1:N
                    for j in 1:N
```

```
                        Tijt[i, j, m] = logsumexp([lps(a[m,n], TijT[i, j, n], b[n,Γ]) for n in 1:N])
                        i==m && (Tijt[i, j, m] = logaddexp(Tijt[i, j, m], lps(βoi_T[j], a[m,j], b[j, Γ])))
                    end
                    for γ in 1:B
                        Eit[γ, i, m] = logsumexp([lps(b[n,Γ], a[m,n], EiT[γ, i, n]) for n in 1:N])
                        i==m && γ==γt && (Eit[γ, i, m] = logaddexp(Eit[γ, i, m], βoi_t[m]))
                    end
                end
            end
            βoi_T = βoi_t; TijT = Tijt; EiT = Eit
        end

        #TERMINATION
        Γ = observations[1,o]
        α1oi[o,:] = [lps(a[i], b[i, Γ]) for i in 1:N]
        β1oi[o,:] = βoi_T
        log_pobs[o] = logsumexp(lps.(α1oi[o,:], βoi_T[:]))
        Eoi[o,:,:] = [logsumexp([lps(EiT[γ,i,m], a[m], b[m,γt]) for m in 1:N]) for i in 1:N, γ in 1:B]
        Toij[o,:,:] = [logsumexp([lps(TijT[i,j,m], a[m], b[m,γt]) for m in 1:N]) for i in 1:N, j in 1:N]
    end

    #INTEGRATE ACROSS OBSERVATIONS AND SOLVE FOR NEW BHMM PARAMS
    obs_penalty=log(O)
    a_o=α1oi.+β1oi•-logsumexp.(eachrow(α1oi.+β1oi))
    new_a=logsumexp.(eachcol(a_o))•-obs_penalty

    a_int = Toij•-logsumexp.([Toij[o,i,:] for o in 1:O, i in 1:N])
    new_a = logsumexp.([a_int[:,i,j] for i in 1:N, j in 1:N])•-obs_penalty

    e_int=Eoi•-logsumexp.([Eoi[o,j,:] for o in 1:O, j in 1:N])
    new_b=logsumexp.([e_int[:,j,d] for d in 1:B, j in 1:N])•-obs_penalty
    new_D=[Categorical(exp.(new_b[:,i])) for i in 1:N]

    return typeof(hmm)(exp.(new_a), exp.(new_a), new_D), lps(log_pobs)
end
```

```julia

#HMMBase
function mouchet_mle_step(hmm::BHMM, observations) where F
    # NOTE: This function works but there is room for improvement.

    log_likelihoods = mouchet_log_likelihoods(hmm, observations)

    log_α = mouchet_messages_forwards_log(hmm.a, hmm.A, log_likelihoods)
    log_β = mouchet_messages_backwards_log(hmm.A, log_likelihoods)
    log_A = log.(hmm.A)

    normalizer = logsumexp(log_α[1,:] + log_β[1,:])

    # E-step

    T, K = size(log_likelihoods)
    log_ξ = zeros(T, K, K)

    @inbounds for t = 1:T-1, i = 1:K, j = 1:K
        log_ξ[t,i,j] = log_α[t,i] + log_A[i,j] + log_β[t+1,j] + log_likelihoods[t+1,j] - normalizer
    end

    ξ = exp.(log_ξ)
    ξ ./= sum(ξ, dims=[2,3])

    # M-step
    new_A = sum(ξ[1:end-1,:,:], dims=1)[1,:,:] #index fix-MM
    new_A ./= sum(new_A, dims=2)

    new_a = exp.((log_α[1,:] + log_β[1,:]) .- normalizer)
    new_a ./= sum(new_a)

    # TODO: Cleanup/optimize this part
    γ = exp.((log_α .+ log_β) .- normalizer)

    B = Categorical[]
    for (i, d) in enumerate(hmm.B)
        # Super hacky ...
        # https://github.com/JuliaStats/Distributions.jl/issues/809
        push!(B, fit_mle(Categorical, permutedims(observations), γ[:,i]))
    end

```

```
    typeof(hmm)(new_a, new_A, B), normalizer
end


function mouchet_log_likelihoods(hmm, observations)
    hcat(map(d -> logpdf.(d, observations), hmm.B)...)
end



function mouchet_messages_forwards_log(init_distn, trans_matrix, log_likelihoods)
    # OPTIMIZE
    log_alphas = zeros(size(log_likelihoods))
    log_trans_matrix = log.(trans_matrix)
    log_alphas[1,:] = log.(init_distn) .+ log_likelihoods[1,:]
    @inbounds for t = 2:size(log_alphas)[1]
        for i in 1:size(log_alphas)[2]
            log_alphas[t,i] = logsumexp(log_alphas[t-1,:] .+ log_trans_matrix[:,i]) + log_likelihoods[t,i]
        end
    end
    log_alphas
end

function mouchet_messages_backwards_log(trans_matrix, log_likelihoods)
    # OPTIMIZE
    log_betas = zeros(size(log_likelihoods))
    log_trans_matrix = log.(trans_matrix)
    @inbounds for t = size(log_betas)[1]-1:-1:1
        tmp = view(log_betas, t+1, :) .+ view(log_likelihoods, t+1, :)
        @inbounds for i in 1:size(log_betas)[2]
            log_betas[t,i] = logsumexp(view(log_trans_matrix, i, :) .+ tmp)
        end
    end
    log_betas
end
```

15.2.27 /test/runtests.jl

```
using BioBackgroundModels,BioSequences,DataFrames,Distributed,Distributions,FASTX,Progr
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!
import Serialization: deserialize
include("synthetic_sequence_gen.jl")
include("ref_fns.jl")

#JOB FILEPATHS
#GFF3 feature database, FASTA genome and index paths
Sys.islinux() ? genome = (@__DIR__) * "/synthetic.fna" : genome = (@__DIR__) * "\\synthetic.fna"
Sys.islinux() ? index = (@__DIR__) * "/synthetic.fna.fai" : index = (@__DIR__) * "\\synthetic.fna.fai"
Sys.islinux() ? gff = (@__DIR__) * "/synthetic.gff3" : gff = (@__DIR__) * "\\synthetic.gff3"
Sys.islinux() ? posfasta = (@__DIR__) * "/syntheticpos.fa" : posfasta = (@__DIR__) * "\\syntheticpos.fa"

!isfile(genome) && print_synthetic_fasta(genome)
!isfile(index) && print_synthetic_index(index)
!isfile(gff) && print_synthetic_gff(gff)
!isfile(posfasta) && print_synthetic_position(posfasta)

Random.seed!(1)

@testset "BHMM/BHMM.jl constructors" begin
    good_a = [.20,.30,.20,.30]
    bad_probvec_a = [.25,.25,.25,.27]
    good_A = fill(.25,4,4)
    bad_probvec_A = deepcopy(good_A)
    bad_probvec_A[:,3]•=.27
```

```
    bad_size_A = fill(.25,5,4)
    good_D = Categorical([.25,.25,.25,.25])
    bad_size_D = Categorical([.2,.2,.2,.2,.2])
    good_Dvec = [good_D for i in 1:4]
    bad_size_Dvec =[good_D, good_D, good_D, bad_size_D]
    bad_length_Dvec=[good_D for i in 1:5]

    @test_throws ArgumentError BHMM(bad_probvec_a, good_A, good_Dvec)
    @test_throws ArgumentError BHMM(good_a, bad_probvec_A, good_Dvec)
    @test_throws ArgumentError BHMM(good_a, bad_size_A, good_Dvec)
    @test_throws ArgumentError BHMM(good_a, good_A, bad_size_Dvec)
    @test_throws ArgumentError BHMM(good_a, good_A, bad_length_Dvec)

    hmm=BHMM(good_a, good_A, good_Dvec)
    @test typeof(hmm)==BHMM{Float64}
    @test hmm.a==good_a
    @test hmm.A==good_A
    @test hmm.B==good_Dvec
    @test size(hmm) == (4,4)

    hmm_copy=copy(hmm)
    @test hmm.a==hmm_copy.a
    @test hmm.A==hmm_copy.A
    @test hmm.B==hmm_copy.B

    hmm2=BHMM(good_A, good_Dvec)
    @test hmm2.a==[.25,.25,.25,.25]
    @test hmm2.A==good_A
    @test hmm2.B==good_Dvec
end

@testset "BHMM/chain.jl Chain_ID and EM_step constructors" begin
    obs_id="test"
    K,zero_k,neg_K=4,0,-1
    order,badorder=0,-1
    replicate,zero_rep,neg_rep=1,0,-1

    @test_throws ArgumentError Chain_ID(obs_id, zero_k, order, replicate)
    @test_throws ArgumentError Chain_ID(obs_id, neg_K, order, replicate)
    @test_throws ArgumentError Chain_ID(obs_id, K, badorder, replicate)
    @test_throws ArgumentError Chain_ID(obs_id, K, order, zero_rep)
    @test_throws ArgumentError Chain_ID(obs_id, K, order, neg_rep)

```

```
    id=Chain_ID(obs_id, K, order, replicate)
    @test typeof(id)==Chain_ID
    @test id.obs_id==obs_id
    @test id.K==K
    @test id.order==order
    @test id.replicate==replicate

    good_A = fill(.25,4,4)
    good_D = Categorical([.25,.25,.25,.25])
    good_Dvec = [good_D for i in 1:4]
    hmm=BHMM(good_A,good_Dvec)
    log_p=-.001

    @test_throws ArgumentError EM_step(0,hmm,0.0,0.0,true)
    @test_throws ArgumentError EM_step(-1,hmm,0.0,0.0,true)
    @test_throws ArgumentError EM_step(1,hmm,1.0,0.0,true)

    steppe=EM_step(1,hmm,log_p,0.0,false)
    @test typeof(steppe)==EM_step
    @test steppe.iterate==1
    @test steppe.hmm==hmm
    @test steppe.log_p==log_p
    @test steppe.delta==0.0
    @test steppe.converged==false
end

@testset "utilities/log_prob_sum.jl Log probability summation functions"
↪ begin
    @test isapprox(lps([.1, .1, .1]), .3)
    @test lps([-Inf, .1, .1]) == -Inf
    @test isapprox(lps(.1, .1, .1), .3)
    @test lps(-Inf, .1, .1) == -Inf
    @test lps(.1, -Inf, .1) == -Inf
end

@testset "utilities/HMM_init.jl initialization functions" begin
    K=4
    order=2
    hmm=autotransition_init(K,order)
    @test typeof(hmm)==BHMM{Float64}
    @test size(hmm)==(4,64)
end
```

```
@testset "utilities/load_balancer.jl EM_converge load balancing structs and functions" begin
    @test_throws ArgumentError LoadConfig(0:5,1:5)
    @test_throws ArgumentError LoadConfig(1:5,-1:5)


    job_ids=[Chain_ID("test",6,2,1),Chain_ID("test",4,2,1),Chain_ID("test",2,0,1)

    obs_sets=Dict("test"⇒[LongSequence{DNAAlphabet{2}}("AAAAAAAAAAAAAA")])

    K_exclusive_config=LoadConfig(1:1,0:0)
    O_exclusive_config=LoadConfig(1:6,3:4)
    blacklist_config=LoadConfig(1:6,0:2,blacklist=job_ids)

    whitelist_config=LoadConfig(1:6,0:2,whitelist=[Chain_ID("test",6,2,2)])
    high_config=LoadConfig(4:6,0:2,whitelist=job_ids)

    no_input_hmms, chains, input_channel, output_channel = setup_EM_jobs!(job_ids,obs_sets)
    @test load_balancer(no_input_hmms, input_channel, K_exclusive_config, 3) == (0,0,0,0,0)

    no_input_hmms, chains, input_channel, output_channel = setup_EM_jobs!(job_ids,obs_sets)
    @test load_balancer(no_input_hmms, input_channel, O_exclusive_config, 3) == (0,0,0,0,0)

    no_input_hmms, chains, input_channel, output_channel = setup_EM_jobs!(job_ids,obs_sets)
    @test load_balancer(no_input_hmms, input_channel, blacklist_config, 3) == (0,0,0,0,0)

    no_input_hmms, chains, input_channel, output_channel = setup_EM_jobs!(job_ids,obs_sets)
    @test load_balancer(no_input_hmms, input_channel, whitelist_config, 3) == (0,0,0,0,0)

    no_input_hmms, chains, input_channel, output_channel = setup_EM_jobs!(job_ids,obs_sets)
    high_set=load_balancer(no_input_hmms, input_channel, high_config)

    @test high_set[1]==Chain_ID("test",6,2,1)
    @test high_set[2]==1
```

```
    @test typeof(high_set[3])==BHMM{Float64}
    @test high_set[4]==obs_lh_given_hmm(high_set[5],high_set[3],linear=false)
    @test typeof(high_set[5])==Matrix{UInt8}
end

@testset "utilities/observation_coding.jl Order coding structs and functions" begin
    compound_alphabet=CompoundAlphabet(ACGT, 2)
    @test typeof(compound_alphabet.symbols)==Dict{Mer{DNAAlphabet{2}},Integer}
    @test length(compound_alphabet.symbols)==64
    @test compound_alphabet.symbols[Mer{DNAAlphabet{2}}("AAA")]==1
    @test compound_alphabet.symbols[Mer{DNAAlphabet{2}}("TTT")]==64

    test_seqs = [BioSequences.LongSequence{DNAAlphabet{2}}("ACGTACGTACGTACGT"),BioSequences.L
    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 0 0 0 0 0 0 0 0 0 0]
    target2= [37 58 15 20 37 58 15 20 37 58 15 20 37 58 0; 64 64 64 64 64 0 0 0 0 0 0 0 0 0 0]

    order0_seqs = get_order_n_seqs(test_seqs,0)
    @test typeof(order0_seqs)==BioBackgroundModels.N_Order_ntSequence
    for (mer, code) in CompoundAlphabet(ACGT,0).symbols
        @test order0_seqs.alphabet.symbols[mer]==code
    end
    @test order0_seqs.seq_lengths==[16,7]
    @test order0_seqs.order_kmers[1][1]==Mer{DNAAlphabet{2}}("A")
    code0_seqs = code_seqs(order0_seqs)
    @test target0 == code0_seqs

    order2_seqs = get_order_n_seqs(test_seqs,2)
    @test typeof(order2_seqs)==BioBackgroundModels.N_Order_ntSequence
    for (mer, code) in CompoundAlphabet(ACGT,2).symbols
        @test order2_seqs.alphabet.symbols[mer]==code
    end
    @test order2_seqs.seq_lengths==[16,7]
    @test order2_seqs.order_kmers[1][1]==Mer{DNAAlphabet{2}}("ACG")
    code2_seqs = code_seqs(order2_seqs)
    @test target2 == code2_seqs

    bw_o0_seqs = get_order_n_seqs(test_seqs[end:-1:1], 0)
```

```
    sorted_code_seqs = code_seqs(bw_o0_seqs, sorted=true)
    @test target0 == sorted_code_seqs

    obs_sets=Dict("test1"⇒test_seqs,
                  "test2"⇒test_seqs)

    job_ids=[Chain_ID("test1",4,0,1),Chain_ID("test2",4,0,1),Chain_ID("test1",2,2

    code_dict=code_job_obs(job_ids,obs_sets)
    @test ("test1",0) in keys(code_dict)
    @test ("test1",2) in keys(code_dict)
    @test ("test2",0) in keys(code_dict)
    @test !(("test2",2) in keys(code_dict))
    @test code_dict[("test1",0)]==target0
    @test code_dict[("test1",2)]==target2
    @test code_dict[("test2",0)]==target0

    #test type selection for high order numbers
    o5_nos=get_order_n_seqs(test_seqs,5)
    o5_codes=code_seqs(o5_nos)
    @test typeof(o5_codes)==Matrix{UInt16}

    o7_nos=get_order_n_seqs(test_seqs,7)
    o7_codes=code_seqs(o7_nos)
    @test typeof(o7_codes)==Matrix{UInt32}

end

@testset "genome_sampling/partition_masker.jl functions" begin
    synthetic_seq = BioSequences.LongSequence{DNAAlphabet{2}}(generate_synthetic_seq())
    position_start = 501
    position_length=141
    position_pad=350
    perigenic_pad = 250

    position_df = make_padded_df(posfasta, gff, genome, index, position_pad)
    add_partition_masks!(position_df, gff, perigenic_pad)

    @test position_df.SeqID[1]=="1"
```

```
    @test length(position_df.PadSeq[1]) == position_pad+position_length
    ↪ ==
    ↪ length(position_df.PadStart[1]:position_df.PadStart[1]+position_length+positi
    ↪ == position_df.End[1]-position_df.Start[1]+1+position_pad ==
    ↪ size(position_df.MaskMatrix[1])[1]
    @test
    ↪ synthetic_seq[position_df.PadStart[1]:position_df.End[1]]==position_df.PadSeq

    mm = position_df.MaskMatrix[1]
    idx=1#pos 151
    @test sum(mm[idx:idx+99,1] .== 1)==length(mm[idx:idx+99,1])
    idx+=100#pos 251
    @test sum(mm[idx:idx+259,1] .== 2)==length(mm[idx:idx+259,1])
    idx+=260 #pos 511
    @test sum(mm[idx:idx+59,1] .== 3)==length(mm[idx:idx+59,1])
    idx+=60#pos 571
    @test sum(mm[idx:idx+29,1] .== 2)==length(mm[idx:idx+29,1])
    idx+=30#pos601
    @test sum(mm[idx:idx+40,1] .== 3)==length(mm[idx:idx+40,1])

    #test exception
    position_df = make_padded_df(posfasta, gff, genome, index,
    ↪ position_pad)

    partition_coords_dict=partition_genome_coordinates(gff,
    ↪ perigenic_pad)

    ↪ partitioned_scaffolds=divide_partitions_by_scaffold(partition_coords_dict)
    scaffold_coords_dict = Dict{String,DataFrame}()
    scaffold="1"
    for ((partition, part_scaffold), df) in partitioned_scaffolds
        if scaffold == part_scaffold
            scaffold_coords_dict[partition] = df
        end
    end

    @test_throws DomainError
    ↪ mask_sequence_by_partition(1001,151,scaffold_coords_dict)
end

@testset "genome_sampling/sequence_sampler.jl functions &
↪ API/genome_sampling.jl interface" begin
    #rectifier tests
```

```
    @test rectify_identifier("1")=="1"

    partitions = 3 #exonic, periexonic, intragenic
    sample_set_length=100
    min_sample_window=5
    max_sample_window=25
    perigenic_pad=250
    syn_intergenic_starts = [1]
    syn_intergenic_ends = [250]
    syn_periexonic_starts = [251,571,661,761,861,961]
    syn_periexonic_ends = [510,600,700,800,900,1000]
    syn_periexonic_strands = ['-','-','-','-','-','-']
    syn_exonic_starts = [511,601,701,801,901]
    syn_exonic_ends = [570,660,760,860,960]
    syn_exonic_strands = ['+','-','-','-','-']
    partition_lengths = Dict("exon"=>5*60,"intergenic"=>250,"periexonic"=>450)

    # "Testing sequence sampler fns..."
    synthetic_seq = BioSequences.LongSequence{DNAAlphabet{2}}(generate_synthetic_seq())

    # "Partitioning synthetic genome coordinates..."
    coordinate_partitions = partition_genome_coordinates(gff, perigenic_pad)

    @test coordinate_partitions["intergenic"].Start == syn_intergenic_starts
    @test coordinate_partitions["intergenic"].End == syn_intergenic_ends

    @test coordinate_partitions["periexonic"].Start == syn_periexonic_starts
    @test coordinate_partitions["periexonic"].End == syn_periexonic_ends
    @test coordinate_partitions["periexonic"].Strand == syn_periexonic_strands

    @test coordinate_partitions["exon"].Start == syn_exonic_starts
    @test coordinate_partitions["exon"].End == syn_exonic_ends
    @test coordinate_partitions["exon"].Strand == syn_exonic_strands

    # "Checking sampling functions at all synthetic indices..."

```

```
    input_test_channel, completed_test_channel, progress_channel, setlength = setup_sample_jobs(genome, index, gff, sample_set_length, min_sample_window, max_sample_window, perigenic_pad; deterministic=true)
    while isready(input_test_channel)
        genome_path, genome_index_path, partition_df, partitionid, sample_set_length, sample_window_min, sample_window_max, deterministic = take!(input_test_channel)

        for feature in eachrow(partition_df)
            seqid, scaffold_start, scaffold_end = meta_to_feature_coord(feature.MetaStart, feature.MetaEnd, partition_df)
            @test scaffold_start == feature.Start && scaffold_end == feature.End
        end

        stranded = get_strand_dict()[partitionid]
        @test typeof(stranded) == Bool

        scaffold_sequence_record_dict = build_scaffold_seq_dict(genome, index)
        @test scaffold_sequence_record_dict["1"] == synthetic_seq

        partition_length = partition_df.MetaEnd[end]
        @test partition_length == partition_lengths[partitionid]

        metacoordinate_bitarray = trues(partition_df.MetaEnd[end])

        for bitindex in findall(metacoordinate_bitarray)
            feature_metaStart, feature_metaEnd, strand = get_feature_params_from_metacoord(bitindex, partition_df, stranded)
            @test 1 ≤ feature_metaStart < feature_metaEnd ≤ length(metacoordinate_bitarray)
            @test feature_metaStart in partition_df.MetaStart
            @test feature_metaEnd in partition_df.MetaEnd
            if partitionid == "periexonic"
                @test strand == '-'
            elseif partitionid == "exon"
                @test strand in ['-','+']
            end
            feature_length = length(feature_metaStart:feature_metaEnd)
```

```
            window = determine_sample_window(feature_metaStart, feature_metaEnd, bitindex, metacoordinate_bitarray, sample_window_min, sample_window_max) #get an appropriate sampling window around the selected index, given the feature boundaries and params
            @test 1 ≤ feature_metaStart ≤ window[1] < window[1]+sample_window_min-1 < window[1]+sample_window_max-1≤ window[2] ≤ feature_metaEnd ≤ length(metacoordinate_bitarray)
            sample_scaffoldid, sample_scaffold_start, sample_scaffold_end = meta_to_feature_coord(window[1],window[2],partition_df)
            @test sample_scaffoldid == "1"
            @test 1 ≤ sample_scaffold_start ≤ sample_scaffold_start+sample_window_min ≤ sample_scaffold_end ≤ min(sample_scaffold_start+sample_window_max,1000) ≤ 1000

            strand == '-' ? target_seq=reverse_complement(synthetic_seq[sample_scaffold_start:sam
            :
                target_seq = synthetic_seq[sample_scaffold_start:sample_scaffold_end]

            proposal_sequence = fetch_sequence(sample_scaffoldid, scaffold_sequence_record_dict, sample_scaffold_start, sample_scaffold_end, strand; deterministic=deterministic) #get the sequence associated with the sample window

            @test proposal_sequence == target_seq
        end
    end

    # "Verifying sampling channels..."

    input_sample_channel, completed_sample_channel, progress_channel, setlength = setup_sample_jobs(genome, index, gff, sample_set_length, min_sample_window, max_sample_window, perigenic_pad; deterministic=true)
    get_sample_set(input_sample_channel, completed_sample_channel, progress_channel)

    #collect sample dfs by partition id when ready
    collected_counter = 0
```

```
    sample_record_dfs = Dict{String,DataFrame}()
    while collected_counter < partitions
        wait(completed_sample_channel)
        partition_id, sample_df = take!(completed_sample_channel)
        sample_record_dfs[partition_id] = sample_df
        collected_counter += 1
    end

    #get_sample entire feature selection
    coordinate_partitions = partition_genome_coordinates(gff, perigenic_pad)
    partition_id="exon"
    partition=coordinate_partitions[partition_id]
    add_metacoordinates!(partition)
    metacoordinate_bitarray=trues(partition.MetaEnd[end])
    scaffold_sequence_record_dict = build_scaffold_seq_dict(genome, index)

    #test fetch_sequence strand char ArgumentError
    @test_throws ArgumentError fetch_sequence("1", Dict{String,LongSequence}("1"⇒LongSequence{DNAAlphabet{2}}("AAAAAAAAAAAAA")), 1, 5, 'L')

    #test get_feature_row_index ArgumentError
    @test_throws DomainError get_feature_row_index(partition,0)

    @test length(get_sample(metacoordinate_bitarray, 1, 250, partition, scaffold_sequence_record_dict)[6])==60

    synthetic_seq = BioSequences.LongSequence{DNAAlphabet{2}}(generate_synthetic_seq())
    for (partid, df) in sample_record_dfs
        for sample in eachrow(df)
            target_seq = synthetic_seq[sample.SampleStart:sample.SampleEnd]
            strand = sample.Strand
            if sample.Strand == '-'
                target_seq = reverse_complement(target_seq)
            end
            @test sample.SampleSequence == target_seq
        end
    end

```

```
    #execute_sample_jobs API
    wkpool=addprocs(1)
    @everywhere using BioBackgroundModels

    channels = setup_sample_jobs(genome, index, gff, sample_set_length,
    ↪  min_sample_window, max_sample_window, perigenic_pad;
    ↪  deterministic=true)
    records=execute_sample_jobs(channels, wkpool)

    rmprocs(wkpool)
end

@testset "likelihood_funcs/hmm.jl & bg_lh_matrix.jl functions" begin
    pvec = [.4,.3,.2,.1]
    trans = ones(1,1)
    B = [Categorical(pvec)]
    hmm = BHMM(trans, B)

    testseq=zeros(Int64,1,5)
    testseq[1:4] = [1,2,3,4]
    @test isapprox(get_BGHMM_symbol_lh(testseq, hmm)[1], log.(pvec[1]))
    @test
    ↪  isapprox(obs_lh_given_hmm(testseq,hmm),obs_lh_given_hmm(testseq,hmm,linear=fa
    @test isapprox(sum(get_BGHMM_symbol_lh(testseq,hmm)),
    ↪  obs_lh_given_hmm(testseq,hmm))
    @test isapprox(sum(get_BGHMM_symbol_lh(testseq,hmm)),
    ↪  obs_lh_given_hmm(testseq,hmm,linear=false))

    pvec=[.25,.25,.25,.25]
    trans = [.9 .1
         .1 .9]
    B = [Categorical(pvec), Categorical(pvec)]
    hmm = BHMM(trans, B)

    @test
    ↪  isapprox(obs_lh_given_hmm(testseq,hmm),obs_lh_given_hmm(testseq,hmm,linear=fa
    @test isapprox(sum(get_BGHMM_symbol_lh(testseq,hmm)),
    ↪  obs_lh_given_hmm(testseq,hmm))
    @test isapprox(sum(get_BGHMM_symbol_lh(testseq,hmm)),
    ↪  obs_lh_given_hmm(testseq,hmm,linear=false))

    testseq=zeros(Int64,1,1001)
    testseq[1,1:1000]=rand(1:4,1000)
```

```

    @test isapprox(sum(get_BGHMM_symbol_lh(testseq, hmm)),obs_lh_given_hmm(testseq,hmm))

    Dex = [Categorical([.3, .1, .3, .3]),Categorical([.15, .35, .35, .15])]
    Dper = [Categorical([.15, .35, .35, .15]),Categorical([.4, .1, .1, .4])]
    Dint = [Categorical([.4, .1, .1, .4]),Categorical([.45, .05, .05, .45])]
    BGHMM_dict = Dict{String,BHMM}()
    BGHMM_dict["exon"] = BHMM(trans, Dex)
    BGHMM_dict["periexonic"] = BHMM(trans, Dper)
    BGHMM_dict["intergenic"] = BHMM(trans, Dint)

    position_length=141;perigenic_pad=250;
    position_df = make_padded_df(posfasta, gff, genome, index, position_length)

    reader=open(FASTA.Reader, genome, index=index)
    seq=FASTA.sequence(reader["CM002885.2.1"])
    @test position_df.Start[1]==501
    @test position_df.End[1]==641
    @test position_df.PadSeq[1]==seq[360:641]
    @test position_df.PadStart[1]==360
    @test position_df.RelStart[1]==141
    @test position_df.SeqOffset[1]==0

    offset_position_df = make_padded_df(posfasta, gff, genome, index,600)
    @test offset_position_df.Start[1]==501
    @test offset_position_df.End[1]==641
    @test offset_position_df.PadSeq[1]==seq[1:641]
    @test offset_position_df.PadStart[1]==1
    @test offset_position_df.RelStart[1]==500
    @test offset_position_df.SeqOffset[1]==100

    add_partition_masks!(position_df, gff, perigenic_pad)
    @test all(position_df.MaskMatrix[1][1:151,1].==2)
    @test all(position_df.MaskMatrix[1][152:211,1].==3)
    @test all(position_df.MaskMatrix[1][212:241,1].==2)
    @test all(position_df.MaskMatrix[1][242:end,1].==3)
    @test all(position_df.MaskMatrix[1][1:151,2].==-1)
    @test all(position_df.MaskMatrix[1][152:211,2].==1)
```

```
    @test all(position_df.MaskMatrix[1][212:end,2].==-1)

    lh_matrix=BGHMM_likelihood_calc(position_df,BGHMM_dict)
    @test size(lh_matrix)==(position_length*2,1)


    periexonic_frag=reverse_complement!(LongSequence{DNAAlphabet{2}}(seq[360:510]
    pno=get_order_n_seqs([periexonic_frag],0)
    pcode=code_seqs(pno)
    plh=get_BGHMM_symbol_lh(pcode, BGHMM_dict["periexonic"])
    @test isapprox(lh_matrix[1:151,1], reverse(plh))

    exonic_frag=LongSequence{DNAAlphabet{2}}(seq[511:570])
    eno=get_order_n_seqs([exonic_frag],0)
    ecode=code_seqs(eno)
    elh=get_BGHMM_symbol_lh(ecode, BGHMM_dict["exon"])
    @test isapprox(lh_matrix[152:211,1],elh)
end

@testset "EM/baum-welch.jl MS_HMMBase equivalency and functions" begin
    A = [.5 .5
         .5 .5]
    B = [Categorical(ones(4)/4), Categorical([.7,.1,.1,.1])]
    hmm = BHMM(A, B)
    log_A = log.(hmm.A)

    obs = zeros(UInt8, 22,1)
    obs[1:21] = [4,3,3,2,3,2,1,1,2,3,3,3,4,4,2,3,2,3,4,3,2]
    obs_lengths=[21]

    lls = BioBackgroundModels.bw_llhs(hmm,obs)
    m_lls = mouchet_log_likelihoods(hmm,obs[1:21])

    K,Tmaxplus1,O = size(lls)
    T=Tmaxplus1-1

    #TEST LOG LIKELIHOOD FN
    @test transpose(lls[:,1:end-1,1]) == m_lls

    log_α = BioBackgroundModels.messages_forwards_log(hmm.a, hmm.A, lls,
    obs_lengths)
    m_log_α = mouchet_messages_forwards_log(hmm.a, hmm.A, m_lls)

```

```
    #TEST FORWARD MESSAGES FN
    @test transpose(log_α[:,1:end-1,1]) == m_log_α

    log_β = BioBackgroundModels.messages_backwards_log(hmm.A, lls, obs_lengths)
    m_log_β = mouchet_messages_backwards_log(hmm.A, m_lls)

    #TEST BACKWARDS  MESSAGES FN
    @test isapprox(transpose(log_β[:,1:end-1,1]),  m_log_β)

    lls = permutedims(lls, [2,3,1]) # from (K,T,O) to (T,O,K)
    log_α = permutedims(log_α, [2,3,1])
    log_β = permutedims(log_β, [2,3,1])

    # "Testing obs probability calcs ..."
    #TEST OBSERVATION PROBABILITY CALC
    normalizer = logsumexp(m_log_α[1,:] + m_log_β[1,:])
    log_pobs = logsumexp(lps.(log_α[1,1,:], log_β[1,1,:]))

    @test isapprox(normalizer, log_pobs)

    # "Testing ξ, γ calcs ..."

    log_ξ = fill(-Inf, Tmaxplus1,O,K,K)
    log_γ = fill(-Inf, Tmaxplus1,O,K)

    m_log_ξ = zeros(T, K, K)

    for t = 1:T-1, i = 1:K, j = 1:K
        m_log_ξ[t,i,j] = m_log_α[t,i] + log_A[i,j] + m_log_β[t+1,j] + m_lls[t+1,j] - normalizer
    end

    for i = 1:K, j = 1:K, o = 1:O
        obsl = obs_lengths[o]
        for t = 1:obsl-1 #log_ξ & log_γ calculated to T-1 for each o
           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])
           log_γ[t,o,i] = lps(log_α[t,o,i], log_β[t,o,i], -log_pobs[o])
        end
        t=obsl #log_ξ @ T = 0
        log_ξ[t,o,i,j] = 0
        log_γ[t,o,i] = lps(log_α[t,o,i], log_β[t,o,i], -log_pobs)
```

```
    end

    ξ = exp.(log_ξ)
    k_ξ = sum(ξ, dims=[3,4])
    nan_mask = k_ξ .== 0; k_ξ[nan_mask] ⋅= Inf #prevent NaNs in dummy
    ↪ renorm arising from multiple sequences indexing
    ξ  ./=  k_ξ #dummy renorm across K to keep numerical creep from
    ↪ causing isprobvec to fail on new new_A during hmm creation

    m_ξ = exp.(m_log_ξ)
    m_ξ ./= sum(m_ξ, dims=[2,3])

    #check equivalency of xi calc methods
    @test isapprox(reshape(ξ,Tmaxplus1,K,K)[1:21,:,:],m_ξ)

    γ = exp.(log_γ)
    k_γ = sum(γ, dims=3); k_γ[nan_mask[:,:,:]] ⋅= Inf #prevent NaNs in
    ↪ dummy renorm
    γ ./= k_γ

    m_γ = exp.((m_log_α .+ m_log_β) ⋅- normalizer)

    #check equivalency of gamma calculations
    @test isapprox(reshape(γ,Tmaxplus1,K)[1:21,:,:],m_γ)

    # "Testing initial and transition matrix calcs…"

    m_new_A = sum(m_ξ[1:end-1,:,:], dims=1)[1,:,:]
    m_new_A ./= sum(m_new_A, dims=2)

    new_A = zeros(K,K)
    for i=1:K, j=1:K
        Σotξ_vec = zeros(O)
        Σotγ_vec = zeros(O)
        for o in 1:O
            Σotξ_vec[o] = sum(ξ[1:obs_lengths[o]-1,o,i,j])
            Σotγ_vec[o] = sum(γ[1:obs_lengths[o]-1,o,i])
        end
        new_A[i,j] = sum(Σotξ_vec) / sum(Σotγ_vec)
    end
    new_A ./= sum(new_A, dims=[2]) #dummy renorm

    #check equivalency of transition matrix calculations
```

```
    @test isapprox(m_new_A,new_A)

    m_new_a = exp.((m_log_α[1,:] + m_log_β[1,:]) .- normalizer)
    m_new_a ./= sum(m_new_a)

    new_a = (sum(γ[1,:,:], dims=1)./sum(sum(γ[1,:,:], dims=1)))[1,:]
    new_a ./= sum(new_a) #dummy renorm

    @test isapprox(m_new_a,new_a)

    # "Testing distribution calcs ..."

    F=Univariate
    m_D = Distribution{F}[]
    for (i, d) in enumerate(hmm.B)
        # Super hacky ...
        # https://github.com/JuliaStats/Distributions.jl/issues/809
        push!(m_D, fit_mle(eval(typeof(d).name.name),
         ↪  permutedims(obs[1:21]), m_γ[:,i]))
    end

    obs_mask = .!nan_mask
    obs_collection = obs[obs_mask[:,:]]

    B = Distribution{F}[]
    @inbounds for (i, d) in enumerate(hmm.B)
        # Super hacky ...
        # https://github.com/JuliaStats/Distributions.jl/issues/809
        γ_d = γ[:,:,i]
        push!(B, fit_mle(eval(typeof(d).name.name), obs_collection,
         ↪  γ_d[obs_mask[:,:]]))
        #slowest call by orders of magnitude
    end

    #test maximization equivalency
    for (d, dist) in enumerate(B)
        @test isapprox(dist.support, m_D[d].support)
        @test isapprox(dist.p, m_D[d].p)
    end

    # "Testing mle_step ..."

```

```
    #verify that above methods independently produce equivalent output, and that this is true of multiple identical obs, but not true of different obs sets
    mouchet_hmm = mouchet_mle_step(hmm, obs[1:21])

    new_hmm = BioBackgroundModels.bw_step(hmm, obs, obs_lengths)

    dblobs = zeros(UInt8, 22,2)
    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]
    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]
    dblobs_lengths=[21,21]
    dbl_hmm = BioBackgroundModels.bw_step(hmm, dblobs, dblobs_lengths)

    otherobs = dblobs
    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]
    other_hmm = BioBackgroundModels.bw_step(hmm, otherobs, dblobs_lengths)

    for n in fieldnames(typeof(new_hmm[1]))
        if n == :B
            for (d, dist) in enumerate(new_hmm[1].B)
                @test dist.support == mouchet_hmm[1].B[d].support
                @test isapprox(dist.p,mouchet_hmm[1].B[d].p)
                @test dist.support == dbl_hmm[1].B[d].support
                @test isapprox(dist.p,dbl_hmm[1].B[d].p)
                @test dist.support == other_hmm[1].B[d].support
                @test !isapprox(dist.p, other_hmm[1].B[d].p)
            end
        elseif n == :partition
            @test getfield(new_hmm[1],n) == getfield(mouchet_hmm[1],n)
            @test getfield(new_hmm[1],n) == getfield(dbl_hmm[1],n)
            @test getfield(new_hmm[1],n) == getfield(other_hmm[1],n)
        else
            @test isapprox(getfield(new_hmm[1],n), getfield(mouchet_hmm[1],n))
            @test isapprox(getfield(new_hmm[1],n), getfield(dbl_hmm[1],n))
            @test !isapprox(getfield(new_hmm[1],n), getfield(other_hmm[1],n))
        end
    end

    @test new_hmm[2] == mouchet_hmm[2] ≠ dbl_hmm[2] ≠ other_hmm[2]
```

```
    # "Testing fit_mle! ..."

    # "test fit_mle! function"
    input_hmms= RemoteChannel(()→Channel{Tuple}(1))
    output_hmms = RemoteChannel(()→Channel{Tuple}(30))
    chainid=Chain_ID("Test",2,0,1)
    put!(input_hmms, (chainid, 2, hmm, 0.0, transpose(obs)))
    BioBackgroundModels.EM_converge!(input_hmms, output_hmms, 1;
    ↪  EM_func=BioBackgroundModels.bw_step, max_iterates=4,
    ↪  verbose=true)
    wait(output_hmms)
    workerid, jobid, iterate, hmm3, log_p, epsilon, converged =
    ↪  take!(output_hmms)
    @test jobid == chainid
    @test iterate == 3
    @test assert_hmm(hmm3.a, hmm3.A, hmm3.B)
    @test size(hmm3) == size(hmm) == (2,4)
    @test log_p < 1
    @test log_p == obs_lh_given_hmm(transpose(obs),hmm, linear=false)
    @test converged == false
    wait(output_hmms)
    workerid, jobid, iterate, hmm4, log_p, epsilon, converged =
    ↪  take!(output_hmms)
    @test jobid == chainid
    @test iterate == 4
    @test assert_hmm(hmm4.a, hmm4.A, hmm4.B)
    @test size(hmm4) == size(hmm) == (2,4)
    @test log_p < 1
    @test log_p == obs_lh_given_hmm(transpose(obs),hmm3,linear=false)
    @test converged == false

    # "Test convergence.."
    obs=zeros(UInt8, 101, 2)
    for i in 1:size(obs)[2]
        obs[1:100,i]=rand(1:4,100)
    end
    input_hmms= RemoteChannel(()→Channel{Tuple}(1))
    output_hmms = RemoteChannel(()→Channel{Tuple}(30))
    put!(input_hmms, (chainid, 2, hmm, 0.0, transpose(obs)))
    BioBackgroundModels.EM_converge!(input_hmms, output_hmms, 1;
    ↪  EM_func=BioBackgroundModels.bw_step, delta_thresh=.05,
    ↪  max_iterates=100, verbose=true)
```

```
    wait(output_hmms)
    workerid, jobid, iterate, hmm4, log_p, epsilon, converged = take!(output_hmms)
    while isready(output_hmms)
        workerid, jobid, iterate, hmm4, log_p, epsilon, converged = take!(output_hmms)
    end
    @test converged==1

    #test t_add_categorical_counts DimensionMismatch
    @test_throws DimensionMismatch t_add_categorical_counts!(zeros(4), zeros(UInt8,2,3), zeros(2,2))

end

@testset "EM/churbanov.jl Baum-Welch equivalency and functions" begin
    # "Setting up for MLE function tests.."
    A = fill((1/6),6,6)
    B = [Categorical(ones(4)/4), Categorical([.7,.05,.15,.1]),Categorical([.15,.35,.4,.1]), Categorical([.6,.15,.15,.1]),Categorical([.1,.4,.4,.1]), Categorical([.2,.2,.3,.3])]
    hmm = BHMM(A, B)
    log_A = log.(hmm.A)

    obs = zeros(Int64,1,250)
    obs[1:249] = rand(1:4,249)
    obs_lengths=[249]
    # "Testing mle_step..."

    #verify that above methods independently produce equivalent output, and that this is true of multiple identical obs, but not true of different obs sets
    mouchet_hmm = mouchet_mle_step(hmm, obs[1:249])

    new_hmm = linear_step(hmm, obs, obs_lengths)

    ms_sng = BioBackgroundModels.bw_step(hmm, Array(transpose(obs)), obs_lengths)

    dblobs = zeros(Int64, 2,250)
    dblobs[1,1:249] = obs[1:249]
    dblobs[2,1:249] = obs[1:249]
```

```
    dblobs_lengths=[249,249]
    dbl_hmm = linear_step(hmm, dblobs, dblobs_lengths)


    ms_dbl =BioBackgroundModels.bw_step(hmm,
     ↪  Array(transpose(dblobs)),dblobs_lengths)

    otherobs = deepcopy(dblobs)
    otherobs[2,1:249] = rand(1:4,249)
    if otherobs[1,1]==otherobs[2,1]
        otherobs[1,1]==1&&(otherobs[2,1]=2)
        otherobs[1,1]==2&&(otherobs[2,1]=3)
        otherobs[1,1]==3&&(otherobs[2,1]=4)
        otherobs[1,1]==4&&(otherobs[2,1]=1)
    end

    other_hmm = linear_step(hmm, otherobs, dblobs_lengths)

    ms_hmm = BioBackgroundModels.bw_step(hmm,
     ↪  Array(transpose(otherobs)),dblobs_lengths)

    for n in fieldnames(typeof(new_hmm[1]))
        if n == :B
            for (d, dist) in enumerate(new_hmm[1].B)
            @test dist.support==mouchet_hmm[1].B[d].support
            @test isapprox(dist.p,mouchet_hmm[1].B[d].p)
            @test dist.support==ms_sng[1].B[d].support
            @test isapprox(dist.p,ms_sng[1].B[d].p)
            @test dist.support==dbl_hmm[1].B[d].support
            @test isapprox(dist.p,dbl_hmm[1].B[d].p)
            @test dist.support==ms_dbl[1].B[d].support
            @test isapprox(dist.p,ms_dbl[1].B[d].p)
            @test dist.support==other_hmm[1].B[d].support
            @test ms_hmm[1].B[d].support==other_hmm[1].B[d].support
            @test !isapprox(dist.p, other_hmm[1].B[d].p)
            @test isapprox(ms_hmm[1].B[d].p, other_hmm[1].B[d].p)

            end
        elseif n == :partition
            @test getfield(new_hmm[1],n) == getfield(mouchet_hmm[1],n)
            @test getfield(new_hmm[1],n) == getfield(dbl_hmm[1],n)
            @test getfield(new_hmm[1],n) == getfield(other_hmm[1],n)
        else
```

```
            @test isapprox(getfield(new_hmm[1],n), getfield(mouchet_hmm[1],n))
            @test isapprox(getfield(new_hmm[1],n), getfield(ms_sng[1],n))

            @test isapprox(getfield(new_hmm[1],n), getfield(dbl_hmm[1],n))
            @test isapprox(getfield(new_hmm[1],n), getfield(ms_dbl[1],n))

            @test !isapprox(getfield(new_hmm[1],n), getfield(other_hmm[1],n))
            @test isapprox(getfield(ms_hmm[1],n), getfield(other_hmm[1],n))

        end
    end

    @test ms_sng[2] == new_hmm[2] == mouchet_hmm[2] ≠ dbl_hmm[2] == ms_dbl[2] ≠ other_hmm[2] == ms_hmm[2]

    # "Testing fit_mle! ..."

    #test fit_mle! function
    input_hmms= RemoteChannel(()→Channel{Tuple}(1))
    output_hmms = RemoteChannel(()→Channel{Tuple}(30))
    chainid=Chain_ID("Test",6,0,1)
    put!(input_hmms, (chainid, 2, hmm, 0.0, obs))
    BioBackgroundModels.EM_converge!(input_hmms, output_hmms, 1; max_iterates=4, verbose=true)
    wait(output_hmms)
    workerid, jobid, iterate, hmm3, log_p, delta, converged = take!(output_hmms)
    @test jobid == chainid
    @test iterate == 3
    @test BioBackgroundModels.assert_hmm(hmm3.a, hmm3.A, hmm3.B)
    @test size(hmm3) == size(hmm) == (6,4)
    @test log_p < 1
    @test log_p == linear_likelihood(obs, hmm)
    @test converged == false
    wait(output_hmms)
    workerid, jobid, iterate, hmm4, log_p, delta, converged = take!(output_hmms)
    @test jobid == chainid
    @test iterate == 4
```

```
    @test BioBackgroundModels.assert_hmm(hmm4.a, hmm4.A, hmm4.B)
    @test size(hmm4) == size(hmm) == (6,4)
    @test log_p < 1
    @test log_p == linear_likelihood(obs, hmm3)
    @test converged == false

    # "Test convergence.."
    obs=zeros(Int64, 4, 1001)
    for o in 1:size(obs)[1]
        obsl=rand(100:1000)
        obs[o,1:obsl]=rand(1:4,obsl)
    end
    input_hmms= RemoteChannel(()→Channel{Tuple}(1))
    output_hmms = RemoteChannel(()→Channel{Tuple}(Inf))
    put!(input_hmms, (chainid, 2, hmm, 0.0, obs))
    BioBackgroundModels.EM_converge!(input_hmms, output_hmms, 1;
    ↪ delta_thresh=.05, max_iterates=100, verbose=true)
    wait(output_hmms)
    workerid, jobid, iterate, hmm4, log_p, delta, converged =
    ↪ take!(output_hmms)
    while isready(output_hmms)
        workerid, jobid, iterate, hmm4, log_p, delta, converged =
        ↪ take!(output_hmms)
    end
    @test converged==1
end

@testset "API/EM_master.jl and reports.jl API tests" begin
    #CONSTANTS FOR BHMM LEARNING
    replicates = 4 #repeat optimisation from this many seperately
    ↪ initialised samples from the prior
    Ks = [1,2,4,6] #mosaic class #s to test
    order_nos = [0,1,2] #DNA kmer order #s to test
    sample_set_length=100
    min_sample_window=5
    max_sample_window=25
    perigenic_pad=250

    wkpool=addprocs(2)
    @everywhere using BioBackgroundModels

```

```
    channels = setup_sample_jobs(genome, index, gff, sample_set_length, min_sample_window, max_sample_window, perigenic_pad; deterministic=true)
    sample_record_dfs=execute_sample_jobs(channels, wkpool)
    training_sets, test_sets = split_obs_sets(sample_record_dfs)

    rmprocs(wkpool)


    job_ids=Vector{Chain_ID}()
    for (obs_id, obs) in training_sets, K in Ks, order in order_nos, rep in 1:replicates
        push!(job_ids, Chain_ID(obs_id, K, order, rep))
    end

    no_input_hmms, chains, input_hmms, output_hmms = setup_EM_jobs!(job_ids, training_sets)
    @test no_input_hmms == replicates*length(Ks)*length(order_nos)*length(training_sets)

    while isready(input_hmms)
        jobid, start_iterate, hmm, last_norm, observations = take!(input_hmms)
        @test last_norm == linear_likelihood(observations, hmm)
        obs_lengths = [findfirst(iszero,observations[o,:])-1 for o in 1:size(observations)[1]]
        #make sure input HMMs are valid and try to mle_step them and ensure their 1-step children are valid
        @test assert_hmm(hmm.a, hmm.A, hmm.B)
        new_hmm, prob = linear_step(hmm,observations,obs_lengths)
        @test assert_hmm(hmm.a, hmm.A, hmm.B)
        @test prob < 0
    end

    genome_reader = open(FASTA.Reader, genome, index=index)

    seq=LongSequence{DNAAlphabet{2}}(FASTA.sequence(genome_reader["CM002885.2.1"]
    seqdict = Dict("test"⇒[seq for i in 1:3])
    seqdict["blacklist"]=seqdict["test"]

    job_ids=Vector{Chain_ID}()
    push!(job_ids, Chain_ID("blacklist", 1, 0, 1))
    push!(job_ids, Chain_ID("blacklist", 1, 0, 2))
```

```
    Ks=[1,2]; order_nos=[0,1]
    for K in Ks, order in order_nos, replicate in 1:2
        push!(job_ids, Chain_ID("test", K, order, replicate))
    end

    wkpool=addprocs(2, topology=:master_worker)
    @everywhere using BioBackgroundModels

    load_dict=Dict{Int64,LoadConfig}()
    for (n,wk) in enumerate(wkpool)
        n==1 &&
         ↪  (load_dict[wk]=LoadConfig(1:2,0:1,blacklist=[Chain_ID("blacklist",1,0,1)]
        n==2 && (load_dict[wk]=LoadConfig(1:2,0:1))
    end
    #test work resumption, load configs

    em_jobset=setup_EM_jobs!(job_ids, seqdict, delta_thresh=10000.)
    execute_EM_jobs!(wkpool, em_jobset..., "testchains",
     ↪  delta_thresh=10000., verbose=true, load_dict=load_dict)

    for (chain_id,chain) in em_jobset[2]
        @test chain[end].converged == true
        @test chain[end].delta ≤ 10000
        @test typeof(chain[end].hmm)==BHMM{Float64}
        @test 1 ≤ chain[end].iterate ≤ 5000
    end

    wkpool=addprocs(2, topology=:master_worker)
    @everywhere using BioBackgroundModels

    em_jobset=setup_EM_jobs!(job_ids, seqdict,
     ↪  chains=deserialize("testchains"), delta_thresh=.1)
    execute_EM_jobs!(wkpool, em_jobset..., "testchains", delta_thresh=.1,
     ↪  verbose=true)

    for (chain_id,chain) in em_jobset[2]
        @test chain[end].converged == true
        @test chain[end].delta ≤ .1
        @test typeof(chain[end].hmm)==BHMM{Float64}
        @test 1 ≤ chain[end].iterate ≤ 5000
    end

    #test for already converged warning
```

```
    wkpool=addprocs(2, topology=:master_worker)
    @everywhere using BioBackgroundModels
    @test_throws ArgumentError execute_EM_jobs!(wkpool, em_jobset ... ,
     ↪  "testchains", delta_thresh=.01, verbose=true)

    rm("testchains")

    @test_throws ArgumentError
     ↪  generate_reports(Dict{Chain_ID,Vector{EM_step}}(),seqdict)
    @test_throws ArgumentError
     ↪  generate_reports(em_jobset[2],Dict{String,Vector{LongSequence{DNAAlphabet{2}}

    @test_throws ArgumentError
     ↪  BioBackgroundModels.report_chains(em_jobset[2],Dict{String,Vector{LongSequenc

    report_folders=generate_reports(em_jobset[2],seqdict)
    folder=report_folders["test"]
    @test "test"==folder.partition_id
    show(folder.partition_report)
    show(folder.replicate_report)
    for id in folder.partition_report.best_repset
        show(folder.chain_reports[id])
    end
end

rm(genome)
rm(index)
rm(gff)
rm(posfasta)
```

15.2.28 /test/synthetic_sequence_gen.jl

```
function print_synthetic_fasta(path::String,line_length::Integer=80)
        header=">CM002885.2.1 BioBackgroundModels Synthetic chromosome
         ↪  for tests\n"
        write(path, header,
         ↪  format_lines(generate_synthetic_seq(),line_length))
end

function print_synthetic_index(path::String)
        write(path,
         ↪  "CM002885.2.1        1000        5        80        81\n")
```

```
end

function print_synthetic_position(path::String)
        write(path, ">1 start 501 end 641 smt_pos 111 smt_value 205.11821 fuzziness_score 43.53698\n",generate_synthetic_seq()[501:641],"\n")
end

function print_synthetic_gff(path::String)
gff_txt = "##gff-version 3
##sequence-region   1 1 1000
#!genome-build BGHMM Synthetic
#!genome-version Synth1.0
#!genome-date 2019-08
#!genome-build-accession NCBI:FAKE
#!genebuild-last-updated 2019-08
1	Ensembl	chromosome	1	1000	.	.	.
###
1	ensembl_havana	gene	501	1000	.	-	.	gene [Source:FAKESRC];gene_id=ENSDARG00000000001;logic_name=ensembl_havana_gene;versio
1	ensembl_havana	mRNA	501	1000	.	-	.
1	ensembl_havana	three_prime_UTR	501	510	.
1	ensembl_havana	exon	501	570	.	+	.
1	ensembl_havana	CDS	511	570	.	+	0
1	ensembl_havana	exon	601	660	.	-	.
1	ensembl_havana	CDS	601	660	.	-	1
1	ensembl_havana	exon	701	760	.	-	.
1	ensembl_havana	CDS	701	760	.	-	0
1	ensembl_havana	exon	801	860	.	-	.
1	ensembl_havana	CDS	801	860	.	-	0
1	ensembl_havana	exon	901	975	.	-	.
1	ensembl_havana	CDS	901	960	.	-	1
1	ensembl_havana	five_prime_UTR	961	975	.	-
###\n"
        write(path,gff_txt)
end
```

```
function generate_synthetic_seq(gene_start::Integer=501, UTR3L::Integer=10, exon_length::Integer=60, intron_length::Integer=40, no_exons::Integer=5, UTR5L::Integer=25, OAL::Integer=1000, (iseq,pseq,eseq)::Tuple{String,String,String}=("AT","CG","CAT"); verbose::Bool=false)
        @assert length(iseq) == 2
        @assert length(pseq) == 2
        @assert length(eseq) == 3
        @assert mod(gene_start-1,2) == 0
        @assert mod(exon_length,3) == 0
        @assert mod(intron_length,2) == 0
        gene_length=gene_start+UTR3L+(no_exons*exon_length)+((no_exons-1)*intron_leng
        @assert gene_length ≤ OAL
        seq=""
        for i in 1:((gene_start-1) / 2)
                seq *= iseq
        end
        verbose && @info "Intergenic length $(length(seq))"
        for i in 1:UTR3L / 2
                seq *= pseq
        end
        verbose && @info "Added 3'UTR ... $(length(seq))"
        for ex in 1:no_exons-1
                for i in 1:(exon_length/3)
                        seq*=eseq
                end
                ex == 1 ? ilength = 30 : ilength = 40
                for i in 1:(ilength/2)
                        seq*=pseq
                end
        end
        for i in 1:(exon_length/3)
                seq*=eseq
        end
        verbose && @info "Added exons and introns ... $(length(seq))"
        for i in 1:UTR5L / 2
                seq *= pseq
        end
        verbose && @info "Added 5'UTR ... $(length(seq))"

        for i in 1:((OAL-length(seq))/2)
                seq*=iseq
```

```
        end

        verbose && @info "Generated synthetic genome sequence of length $(length(seq))"

        return seq
end

  function format_lines(seq::String, line_length::Integer)
        a=join((SubString(seq,i,min(i+line_length-1,length(seq))) for i=1:line_length:length(seq)),'\n')
        return(a)
  end
```

15.3 BioMotifInference

15.3.1 /README.md

```
## BioMotifInference
[![Build Status](https://travis-ci.org/mmattocks/BioMotifInference.jl.svg?branch=master)](
[![codecov](https://codecov.io/gh/mmattocks/BioMotifInference.jl/branch/master/graph/
[![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.repostatu
```

15.3.2 /src/BioMotifInference.jl

```
module BioMotifInference
    using BioBackgroundModels, BioSequences, Distributed, Distributions, Serialization, UnicodePlots
    import DataFrames:DataFrame
    import ProgressMeter: AbstractProgress, Progress, @showprogress, next!, move_cursor_up_while_clearing_lines, printover, durationstring
    import Printf: @sprintf
    import StatsFuns: logaddexp, logsumexp #both are needed as logsumexp for two terms is deprecated
    import Random: rand, seed!, shuffle!
    import Distances: euclidean
```

```
9
10    #CONSTANTS AND PERMUTE FUNCTION ARGUMENT DEFAULTS GIVING RISE TO
      ↪  IMPLEMENTATION-SPECIFIC SAMPLING EFFECTS
11    global MOTIF_EXPECT=1. #motif expectation- this value to be divided
      ↪  by obs lengths to obtain penalty factor for scoring
12
13    global REVCOMP=true #calculate scores on both strands?
14
15    global TUNING_MEMORY=20 #coefficient multiplied by Permute_Instruct
      ↪  function call limit to give total number of calls remembered by
      ↪  tuner per function
16    global CONVERGENCE_MEMORY=500 #number of iterates to display for
      ↪  convergence interval history
17
18    global SRC_PERM_FREQ=.5 #frequency with which random_decorrelate will
      ↪  permute a source
19
20    global PWM_SHIFT_DIST=Weibull(.5,.1) #distribution of weight matrix
      ↪  permutation magnitudes
21    global PWM_SHIFT_FREQ=.2 #proportion of positions in source to
      ↪  permute weight matrix
22    global PWM_LENGTHPERM_FREQ=.2 #proportion of sources to permute
      ↪  length
23    global LENGTHPERM_RANGE=1:3
24
25    global PRIOR_WT=3. #estimate prior dirichlets from product of this
      ↪  constant and sample "mle" wm
26    global PRIOR_LENGTH_MASS=.8
27
28    global EROSION_INFO_THRESH=1.
29
30    global CONSOLIDATE_THRESH=.035
31
32    include("IPM/ICA_PWM_Model.jl")
33    export Model_Record
34    export ICA_PWM_Model
35    include("IPM/IPM_likelihood.jl")
36    include("IPM/IPM_prior_utilities.jl")
37    assemble_source_priors
38    include("ensemble/IPM_Ensemble.jl")
39    export IPM_Ensemble, assemble_IPMs
40    include("permutation/permute_utilities.jl")
41    include("permutation/orthogonality_helper.jl")
```

```
    include("permutation/permute_functions.jl")
    export full_perm_funcvec
    include("permutation/permute_control.jl")
    export Permute_Instruct
    include("permutation/Permute_Tuner.jl")
    include("ensemble/ensemble_utilities.jl")
    export ensemble_history, reset_ensemble!, move_ensemble!, copy_ensemble!, rewind_ensemble, complete_evidence, get_model, show_models
    include("utilities/model_display.jl")
    include("utilities/worker_diagnostics.jl")
    include("utilities/ns_progressmeter.jl")
    include("utilities/synthetic_genome.jl")
    include("utilities/worker_sequencer.jl")
    export synthetic_sample
    include("nested_sampler/nested_step.jl")
    include("nested_sampler/converge_ensemble.jl")
    export converge_ensemble!

end # module
```

15.3.3 /src/IPM/ICA_PWM_Model.jl

```
struct Model_Record #record struct to associate a log_Li with a saved, calculated model
    path::String
    log_Li::AbstractFloat
end

struct ICA_PWM_Model #Independent component analysis position weight matrix model
    name::String #designator for saving model to posterior
    origin::String #functions instantiating IPMs should give an informative desc of the means by which the model was generated
    sources::Vector{Tuple{<:AbstractMatrix{<:AbstractFloat},<:Integer}} #vector of PWM signal sources (LOG PROBABILITY!!!) tupled with an index denoting the position of the first PWM base on the prior matrix- allows us to permute length and redraw from the appropriate prior position
    source_length_limits::UnitRange{<:Integer} #min/max source lengths for init and permutation
    mix_matrix::BitMatrix # obs x sources bool matrix
```

```
    log_Li::AbstractFloat
    permute_blacklist::Vector{Function} #blacklist of functions that ought not be used to permute this model (eg. because to do so would not generate a different model for IPMs produced from fitting the mix matrix)
    function ICA_PWM_Model(name, origin, sources, source_length_limits, mix_matrix, log_Li, permute_blacklist=Vector{Function}())
        verify_srcs(name, origin, sources)
        new(name, origin, sources, source_length_limits, mix_matrix, log_Li, permute_blacklist)
    end
end

#ICA_PWM_Model FUNCTIONS
ICA_PWM_Model(name::String, source_priors::AbstractVector{<:Union{<:AbstractVector{<:Dirichlet{<:AbstractFloa
mix_prior::Tuple{BitMatrix,<:AbstractFloat}, bg_scores::AbstractArray{<:AbstractFloat}, observations::AbstractArray{<:Integer}, source_length_limits::UnitRange{<:Integer}) = init_IPM(name, source_priors,mix_prior,bg_scores,observations,source_length_limits)

function verify_srcs(name, ori, sources::Vector{<:Tuple{<:AbstractMatrix{<:AbstractFloat},<:Integer}})
    for (n,(pwm,pi)) in enumerate(sources)
        !all(isprobvec(exp.(pwm[pos,:])) for pos in 1:size(pwm,1)) && throw(DomainError("Bad probvec in model $name, source $n, origin $(ori):$(exp.(pwm))"))
    end
end

#MODEL INIT
function init_IPM(name::String, source_priors::AbstractVector{<:Union{<:AbstractVector{<:Dirichlet{<:AbstractFloa
mix_prior::Tuple{BitMatrix,<:AbstractFloat}, bg_scores::AbstractArray{<:AbstractFloat}, observations::AbstractArray{<:Integer}, source_length_limits::UnitRange{<:Integer})
    assert_obs_bg_compatibility(observations,bg_scores)
    T,O = size(observations)
    S=length(source_priors)
    obs_lengths=[findfirst(iszero,observations[:,o])-1 for o in 1:size(observations)[2]]
```

```
    sources=init_logPWM_sources(source_priors, source_length_limits)
    mix=init_mix_matrix(mix_prior,O,S)
    log_lh = IPM_likelihood(sources, observations, obs_lengths, bg_scores, mix)

   return ICA_PWM_Model(name, "init_IPM", sources, source_length_limits, mix, log_lh)
end
                #init_IPM SUBFUNCS
                function assert_obs_bg_compatibility(obs, bg_scores)
                    T,O=size(obs)
                    t,o=size(bg_scores)
                    O!=o && throw(DomainError("Background scores and observations must have same number of observation columns!"))
                    T!=t+1 && throw(DomainError("Background score array must have the same observation lengths as observations!"))
                end

                function init_logPWM_sources(prior_vector::AbstractVector{<:Union{<:Abstra source_length_limits::UnitRange{<:Integer})
                    srcvec = Vector{Tuple{Matrix{Float64},Int64}}()
                        for prior in prior_vector
                            if typeof(prior)<:AbstractVector{<:Dirichlet{<:AbstractF
                                if length(prior) < source_length_limits[1]
                                    length_dist=DiscreteNonParametric(
                                        [source_length_limits ... ],
                                        [PRIOR_LENGTH_MASS,ones(length(source_len
                                    PWM_length=rand(length_dist)
                                elseif source_length_limits[1] < length(prior) < source_length_limits[end]
                                    length_dist=DiscreteNonParametric(
                                        [source_length_limits ... ],
                                        [
                                            (ones(length(prior)-source_length_lim
```

```
                                        ↪  (ones(source_length_limits[end]-lengt
                                    ]
                                )
                                PWM_length=rand(length_dist)
                            else
                                PWM_length=rand(source_length_limits)
                            end

                            if PWM_length>length(prior)

                                ↪  prior_coord=rand(-(PWM_length-length(prior)):
                            else

                                ↪  prior_coord=rand(1:length(prior)-PWM_length+1
                            end

                            PWM = zeros(PWM_length,4)

                            curr_pos=prior_coord
                            for pos in 1:PWM_length
                                curr_pos < 1 || curr_pos >
                                ↪  length(prior) ?
                                ↪  dirichlet=Dirichlet(ones(4)/4) :
                                    dirichlet=prior[curr_pos]
                                PWM[pos, :] = rand(dirichlet)
                            end
                            push!(srcvec, (log.(PWM), prior_coord))
                            ↪  #push the source PWM to the source
                            ↪  vector with the prior coord idx to
                            ↪  allow drawing from the appropriate
                            ↪  prior dirichlets on permuting source
                            ↪  length
                        elseif prior==false
                            PWM_length=rand(source_length_limits)
                            PWM=zeros(PWM_length,4)
                            dirichlet=Dirichlet(ones(4)/4)
                            for pos in 1:PWM_length
                                PWM[pos,:] = rand(dirichlet)
                            end
                            push!(srcvec, (log.(PWM), 1))
                        else
```

```
                                throw(ArgumentError("Bad prior supplied for ICA_PWM_Model!"))
                            end
                        end
                    return srcvec
                end

                function init_mix_matrix(mix_prior::Tuple{BitMatrix,<:AbstractFloat}, no_observations::Integer, no_sources::Integer)
                    inform,uninform=mix_prior
                    if size(inform,2) > 0
                        @assert size(inform,1)==no_observations && size(inform,2)≤no_sources "Bad informative mix prior dimensions!"
                    end
                    @assert 0.0 ≤ uninform ≤1.0 "Uninformative mix prior not between 0.0 and 1.0!"
                    mix_matrix = falses(no_observations, no_sources)
                    if size(inform,2)>0
                        mix_matrix[:,1:size(inform,2)]=inform
                    end
                    for index in CartesianIndices((1:no_observations,size(inform,2)+1:no_sourc
                        rand() ≤ uninform && (mix_matrix[index] = true)
                    end
                    return mix_matrix
                end

function Base.show(io::IO, m::ICA_PWM_Model; nsrc::Integer=length(m.sources), progress=false)
    nsrc == 0 && (nsrc=length(m.sources))
    nsrc>length(m.sources) && (nsrc=length(m.sources))
    nsrc==length(m.sources) ? (srcstr="All") : (srcstr="Top $nsrc")

    printidxs,printsrcs,printfreqs=sort_sources(m,nsrc)

    printstyled(io, "ICA PWM Model $(m.name) w/ logLi $(m.log_Li)\n", bold=true)
    println(io, srcstr*" sources:")
    for src in 1:nsrc
        print(io, "S$(printidxs[src]), $(printfreqs[src]*100)%: ")
        pwmstr_to_io(io, printsrcs[src])
```

```
        println(io)
    end
    println(m.origin)

    progress && return(nsrc+4)
end

function sort_sources(m, nsrc)
    printidxs=Vector{Integer}()
    printsrcs=Vector{Matrix{Float64}}()
    printfreqs=Vector{Float64}()

    freqs=vec(sum(m.mix_matrix,dims=1)); total=size(m.mix_matrix,1)
    sortfreqs=sort(freqs,rev=true); sortidxs=sortperm(freqs,rev=true)
    for srcidx in 1:nsrc
        push!(printidxs, sortidxs[srcidx])
        push!(printsrcs, m.sources[sortidxs[srcidx]][1])
        push!(printfreqs, sortfreqs[srcidx]/total)
    end
    return printidxs,printsrcs,printfreqs
end
```

15.3.4 /src/IPM/IPM_likelihood.jl

```
#LIKELIHOOD SCORING FUNCS
function IPM_likelihood(sources::AbstractVector{<:Tuple{<:AbstractMatrix{<:AbstractFloat},
observations::AbstractMatrix{<:Integer},
obs_lengths::AbstractVector{<:Integer},
bg_scores::AbstractArray{<:AbstractFloat}, mix::BitMatrix,
revcomp::Bool=REVCOMP, returncache::Bool=false,
cache::AbstractVector{<:AbstractFloat}=zeros(0),
clean::AbstractVector{<:Bool}=Vector(falses(size(observations)[2])))
    source_wmls=[size(source[1])[1] for source in sources]
    O = size(bg_scores)[2]
    source_stops=[obsl-wml+1 for wml in source_wmls, obsl in obs_lengths] #stop scannng th source across the observation as the source reaches the end
    L=maximum(obs_lengths)+1

    obs_src_idxs=mix_pull_idxs(mix) #get vectors of sources emitting in each obs
```

```
9
10      revcomp ? (srcs=[cat(source[1],revcomp_pwm(source[1]),dims=3) for
        ↪  source in sources]; motif_expectations = [((MOTIF_EXPECT/2)/obsl)
        ↪  for obsl in obs_lengths]; mat_dim=2) : (srcs=[source[1] for
        ↪  source in sources]; ; motif_expectations = [(MOTIF_EXPECT/obsl)
        ↪  for obsl 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
11
12      lme_vec=zeros(length(sources))
13
14      obs_lhs=Vector{Vector{Float64}}() #setup likelihood vecs for threaded
        ↪  operation-reassembled on return
15      nt=Threads.nthreads()
16      for t in 1:nt-1
17          push!(obs_lhs,zeros(Int(floor(O/nt))))
18      end
19      push!(obs_lhs, zeros(Int(floor(O/nt)+(O%nt))))
20
21      Threads.@threads for t in 1:nt
22          revcomp && (weavevec=zeros(3))
23          revcomp ? (score_mat=zeros(maximum(source_stops),2)) :
            ↪  (score_mat=zeros(maximum(source_stops)))
24          opt = floor(O/nt) #obs per thread
25          score_matrices=Vector{typeof(score_mat)}(undef, length(sources))
            ↪  #preallocate
26          osi_emitting=Vector{Int64}() #preallocate
27          lh_vec = zeros(L) #preallocated likelihood vector is one position
            ↪  (0 initialiser) longer than the longest obs
28
29          for i in 1:Int(opt+(t==nt)*(O%nt))
30              o=Int(i+(t-1)*opt)
31              if clean[o]
32                  obs_lhs[t][i]=cache[o]
33              else
34                  obsl = obs_lengths[o]
35                  oidxs=obs_src_idxs[o]
36                  mixwmls=source_wmls[oidxs]
37
38                  obs_cardinality = length(oidxs) #the more sources, the
                    ↪  greater the cardinality_penalty
39                  if obs_cardinality > 0
```

```
                    revcomp ? score_sources_ds!(score_mat, score_matrices, view(srcs,oidxs), view(observations,:,o), view(source_stops,oidxs,o)) : score_sources_ss!(score_mat, score_matrices, view(srcs,oidxs), view(observations,:,o), view(source_stops,oidxs,o)) #get scores for this observation


                    lme_vec•=motif_expectations[o]
                    penalty_sum = sum(lme_vec[1:obs_cardinality])
                    penalty_sum > 1. && (penalty_sum=1.)
                    cardinality_penalty=log(1.0-penalty_sum)
                else
                    cardinality_penalty=0.0
                end

                revcomp ? (obs_lhs[t][i]=weave_scores_ds!(weavevec, lh_vec, obsl, view(bg_scores,:,o), score_matrices, oidxs, mixwmls, log(motif_expectations[o]), cardinality_penalty, osi_emitting)) : (obs_lhs[t][i]=weave_scores_ss!(lh_vec, obsl, view(bg_scores,:,o), score_matrices, oidxs, mixwmls, log(motif_expectations[o]), cardinality_penalty, osi_emitting))

                empty!(osi_emitting)

            end
        end
    end

    returncache ? (return lps([lps(obs_lhs[t]) for t in 1:nt]), vcat(obs_lhs ... )) : (return lps([lps(obs_lhs[t]) for t in 1:nt]))
end
                @inline function mix_pull_idxs(A::AbstractArray{Bool})
                    n=count(A)
                    S=[Vector{Int64}() for o in 1:size(A,1)]
                    cnt=1
                    for (i,a) in pairs(A)
                        if a
                            push!(S[i[1]],i[2])
```

```
                        cnt+=1
                    end
                end
                return S
            end

            @inline function revcomp_pwm(pwm::AbstractMatrix{<:AbstractFloat}) #in order to find a motif on the reverse strand, we scan the forward strand with the reverse complement of the pwm, reordered 3' to 5', so that eg. an PWM for an ATG motif would become one for a CAT motif
                return pwm[end:-1:1,end:-1:1]
            end

            @inline function score_sources_ds!(score_mat, score_matrices::AbstractVector{<:AbstractMatrix{<:AbstractFloat}}, sources, observation::AbstractVector{<:Integer}, source_stops)
                for (s,source) in enumerate(sources)
                    for t in 1:source_stops[s]
                        for position in 1:size(source,1)
                            score_loc = t+position-1 #score_loc is the position of the obs to be scored by PWM
                            score_mat[t,1] += source[position,observation[score_loc],1] #add the appropriate log PWM value from the source to the score
                            score_mat[t,2] += source[position,observation[score_loc],2] #add the appropriate log PWM value from the source to the score

                        end
                    end
                    score_matrices[s]=score_mat[1:source_stops[s],:] #copy score matrix to vector
                    score_mat•=0. #reset score matrix
                end
            end

```

```
95          @inline function score_sources_ss!(score_mat,
            ↪ score_matrices::AbstractVector{<:AbstractArray{<:AbstractFloat}},
            ↪ sources, observation::AbstractVector{<:Integer},
            ↪ source_stops)
96              for (s,source) in enumerate(sources)
97                  for t in 1:source_stops[s]
98                      for position in 1:size(source,1)
99                          score_loc = t+position-1 #score_loc is
                            ↪ the position of the obs to be scored
                            ↪ by PWM
100                         score_mat[t] +=
                            ↪ source[position,observation[score_loc]]
                            ↪ #add the appropriate log PWM value
                            ↪ from the source to the score
101
102                     end
103                 end
104                 score_matrices[s]=score_mat[1:source_stops[s]]
                    ↪ #copy score matrix to vector
105                 score_mat•=0. #reset score matrix
106             end
107         end
108
109         @inline function weave_scores_ds!(weavevec, lh_vec,
            ↪ obsl::Integer, bg_scores::SubArray,
            ↪ score_mat::AbstractVector{<:AbstractMatrix{<:AbstractFloat}},
            ↪ obs_source_indices::AbstractVector{<:Integer},
            ↪ source_wmls::AbstractVector{<:Integer},
            ↪ log_motif_expectation::AbstractFloat,
            ↪ cardinality_penalty::AbstractFloat, osi_emitting)
110             for i in 2:obsl+1 #i=1 is ithe lh_vec initializing 0,
                ↪ i=2 is the score of the first background position
                ↪ (ie t=1)
111                 t=i-1
112                 score = lps(lh_vec[i-1], bg_scores[t],
                    ↪ cardinality_penalty)
113
114                 #logic: all observations are scored from t=wml to
                    ↪ the end of the obs-therefore check at each
                    ↪ position for new sources to add (indexed by
                    ↪ vector position to retrieve source wml and
                    ↪ score matrix)
```

```
                if length(osi_emitting)<length(obs_source_indices)
                    for n in 1:length(obs_source_indices)
                        if !(n in osi_emitting)
                            t ≥ source_wmls[n] && (push!(osi_emitting,n))
                        end
                    end
                end

                for n in osi_emitting
                    wml = source_wmls[n]
                    from_score = lh_vec[i-wml+1] #score at the first position of the PWM
                    score_array = score_mat[n] #get the source score matrix
                    score_idx = t - wml + 1 #translate t to score_array idx for emission score
                    f_emit_score = score_array[score_idx,1] #emission score at the last position of the PWM
                    r_emit_score = score_array[score_idx,2]

                    weavevec .= score, lps(from_score, f_emit_score, log_motif_expectation), lps(from_score, r_emit_score, log_motif_expectation)

                    score=logsumexp(weavevec)
                end
                lh_vec[i] = score
            end
            return lh_vec[obsl+1]
        end

        @inline function weave_scores_ss!(lh_vec, obsl::Integer, bg_scores::SubArray, score_mat::AbstractVector{<:AbstractVector{<:AbstractFloat}}, obs_source_indices::AbstractVector{<:Integer}, source_wmls::AbstractVector{<:Integer}, log_motif_expectation::AbstractFloat, cardinality_penalty::AbstractFloat, osi_emitting)
```

```
141                     for i in 2:obsl+1 #i=1 is ithe lh_vec initializing 0,
                         ↪  i=2 is the score of the first background position
                         ↪  (ie t=1)
142                         t=i-1
143                         score = lps(lh_vec[i-1], bg_scores[t],
                             ↪  cardinality_penalty)
144                         #logic: all observations are scored from t=wml to
                             ↪  the end of the obs-therefore check at each
                             ↪  position for new sources to add (indexed by
                             ↪  vector position to retrieve source wml and
                             ↪  score matrix)
145                         if
                             ↪  length(osi_emitting)<length(obs_source_indices)
146                             for n in 1:length(obs_source_indices)
147                                 if !(n in osi_emitting)
148                                     t ⩾ source_wmls[n] &&
                                         ↪  (push!(osi_emitting,n))
149                                 end
150                             end
151                         end
152
153                         for n in osi_emitting
154                             wml = source_wmls[n]
155                             from_score = lh_vec[i-wml+1] #score at the
                                 ↪  first position of the PWM
156                             score_array = score_mat[n] #get the source
                                 ↪  score matrix
157                             score_idx = t - wml + 1 #translate t to
                                 ↪  score_array idx for emission score
158                             f_emit_score = score_array[score_idx]
                                 ↪  #emission score at the last position of
                                 ↪  the PWM
159
160                             score=logaddexp(score, lps(from_score,
                                 ↪  f_emit_score, log_motif_expectation))
161                         end
162                         lh_vec[i] = score
163                     end
164                     return lh_vec[obsl+1]
165                 end
```

15.3.5 /src/IPM/IPM_prior_utilities.jl

```
function read_fa_wms_tr(path::String)
    wms=Vector{Matrix{Float64}}()
    wm=zeros(1,4)
    f=open(path)
    for line in eachline(f)
        prefix=line[1:2]
        prefix == "01" && (wm=transpose([parse(Float64,i) for i in split(line)[2:end]]))
        prefix ≠ "01" && prefix ≠ "NA" && prefix ≠ "PO" && prefix ≠ "//" && (wm=vcat(wm, transpose([parse(Float64,i) for i in split(line)[2:end]])))
        prefix == "//" && push!(wms, wm)
    end
    return wms
end

#wm_samples are in decimal probability space, not log space
function assemble_source_priors(no_sources::Integer, wm_samples::AbstractVector{<:AbstractMatrix{<:AbstractFloat}}, prior_wt::AbstractFloat=PRIOR_WT) #estimate a dirichlet prior on wm_samples inputs; if the number of samples is lower than the number of sources, return a false bool for init and permutation functions
    source_priors = Vector{Union{Vector{Dirichlet{Float64}},Bool}}()
    for source in 1:no_sources
        if source ≤ length(wm_samples)
            push!(source_priors, estimate_dirichlet_prior_on_wm(wm_samples[source], prior_wt))
        else
            push!(source_priors, false)
        end
    end
    return source_priors
end

            function estimate_dirichlet_prior_on_wm(wm::AbstractMatrix{<:AbstractFloat}, wt::AbstractFloat=PRIOR_WT)
                for i in 1:size(wm)[1]
```

```
                    !(isprobvec(wm[i,:])) && throw(DomainError("Bad weight vec supplied to estimate_dirichlet_prior_on_wm! $(wm[i,:])"))
                end
                prior = Vector{Dirichlet{Float64}}()
                for position in 1:size(wm)[1]
                    normvec=wm[position,:]
                    zero_idxs=findall(isequal(0.),wm[position,:])
                    normvec[zero_idxs].+=10^-99
                    push!(prior, Dirichlet(normvec.*wt))
                end
                return prior
            end

function cluster_mix_prior!(df::DataFrame, wms::AbstractVector{<:AbstractMatrix{<:AbstractFloat}})
    mix=falses(size(df,1),length(wms))
    for (o, row) in enumerate(eachrow(df))
        row.cluster ≠ 0 && (mix[o,row.cluster]=true)
    end

    represented_sources=unique(df.cluster)
    return mix[:,represented_sources]
end

function infocenter_wms_trim(wm::AbstractMatrix{<:AbstractFloat}, trimsize::Integer)
    !(size(wm,2)==4) && throw(DomainError("Bad wm! 2nd dimension should be size 4"))
    infovec=get_pwm_info(wm, logsw=false)
    maxval, maxidx=findmax(infovec)
    upstream_extension=Int(floor((trimsize-1)/2))
    downstream_extension=Int(ceil((trimsize-1)/2))
    1+upstream_extension+downstream_extension > size(wm,1) && throw(DomainError("Src too short for trim! $upstream_extension $downstream_extension"))
    return wm[max(1,maxidx-upstream_extension):min(maxidx+downstream_extension,size(wm,1
end

function filter_priors(target_src_no::Integer, target_src_size::Integer, prior_wms::AbstractVector{<:AbstractMatrix{<:AbstractFloat}}, prior_mix::BitMatrix)
```

```
    wms=Vector{Matrix{Float64}}(undef, target_src_no)
    freqsort_idxs=sortperm([sum(prior_mix[:,s]) for s in 1:length(prior_wms)])
    for i in 1:target_src_no
        target_src_idx=freqsort_idxs[i]
        wms[i]=infocenter_wms_trim(prior_wms[target_src_idx], target_src_size)
    end
    return wms
end

function combine_filter_priors(target_src_no::Integer, target_src_size::Integer, prior_wms::Tuple{<:AbstractVector{<:AbstractMatrix{<:AbstractFloat}},AbstractVect
prior_mix::Tuple{BitMatrix,BitMatrix})
    wms=Vector{Matrix{Float64}}(undef, target_src_no)
    cat_wms=vcat(prior_wms[1],prior_wms[2])
    first_freq=[sum(prior_mix[1][:,s]) for s in 1:length(prior_wms[1])]
    second_freq=[sum(prior_mix[2][:,s]) for s in 1:length(prior_wms[2])]
    freqsort_idxs=sortperm(vcat(first_freq,second_freq))
    for i in 1:target_src_no
        target_src_idx=freqsort_idxs[i]
        wms[i]=infocenter_wms_trim(cat_wms[target_src_idx], target_src_size)
    end
    return wms
end
```

15.3.6 /src/ensemble/IPM_Ensemble.jl

```
mutable struct IPM_Ensemble
        path::String #ensemble models and popped-out posterior samples serialised here
        models::Vector{Model_Record} #ensemble keeps paths to serialised models and their likelihood tuples rather than keeping the models in memory

        contour::AbstractFloat#current log_Li[end]
        log_Li::Vector{AbstractFloat} #likelihood of lowest-ranked model at iterate i
        log_Xi::Vector{AbstractFloat} #amt of prior mass included in ensemble contour at Li
```

```
        log_wi::Vector{AbstractFloat} #width of prior mass covered in this iterate
        log_Liwi::Vector{AbstractFloat} #evidentiary weight of the iterate
        log_Zi::Vector{AbstractFloat} #ensemble evidence
        Hi::Vector{AbstractFloat} #ensemble information

        obs_array::Matrix{Integer} #observations Txo
        obs_lengths::Vector{Integer}

        source_priors::AbstractVector{<:Union{<:AbstractVector{<:Dirichlet{<:Abstract #source pwm priors
        mix_prior::Tuple{BitMatrix,AbstractFloat} #prior on %age of observations that any given source contributes to

        bg_scores::Matrix{AbstractFloat} #precalculated background HMM scores, same dims as obs

        sample_posterior::Bool
        posterior_samples::Vector{Model_Record} #list of posterior sample records

        model_counter::Integer

        naive_lh::AbstractFloat #the likelihood of the background model without any sources
end

####IPM_Ensemble FUNCTIONS
IPM_Ensemble(path::String, no_models::Integer, source_priors::AbstractVector{<:Union{<:AbstractVector{<:Dirichlet{<:AbstractFloa mix_prior::Tuple{BitMatrix,<:AbstractFloat}, bg_scores::AbstractMatrix{<:AbstractFloat}, obs::AbstractArray{<:Integer}, source_length_limits; posterior_switch::Bool=false) =
IPM_Ensemble(
        path,
        assemble_IPMs(path, no_models, source_priors, mix_prior, bg_scores, obs, source_length_limits)...,
        [-Inf], #L0 = 0
        [0], #ie exp(0) = all of the prior is covered
        [-Inf], #w0 = 0
        [-Inf], #Liwi0 = 0
```

```
        [-1e300], #Z0 = 0
        [0], #H0 = 0,
        obs,
        [findfirst(iszero,obs[:,o])-1 for o in 1:size(obs)[2]],
        source_priors,
        mix_prior,
        bg_scores, #precalculated background score
        posterior_switch,
        Vector{String}(),
        no_models+1,
        IPM_likelihood(init_logPWM_sources(source_priors,
         ↪  source_length_limits), obs, [findfirst(iszero,obs[:,o])-1 for
         ↪  o in 1:size(obs)[2]], bg_scores,
         ↪  falses(size(obs)[2],length(source_priors))))

IPM_Ensemble(worker_pool::AbstractVector{<:Integer}, path::String,
 ↪  no_models::Integer,
 ↪  source_priors::AbstractVector{<:Union{<:AbstractVector{<:Dirichlet{<:AbstractFloa
 ↪  mix_prior::Tuple{BitMatrix,<:AbstractFloat},
 ↪  bg_scores::AbstractMatrix{<:AbstractFloat}, obs::Array{<:Integer},
 ↪  source_length_limits; posterior_switch::Bool=false) =
IPM_Ensemble(
        path,
        distributed_IPM_assembly(worker_pool, path, no_models,
         ↪  source_priors, mix_prior, bg_scores, obs,
         ↪  source_length_limits)...,
        [-Inf], #L0 = 0
        [0], #ie exp(0) = all of the prior is covered
        [-Inf], #w0 = 0
        [-Inf], #Liwi0 = 0
        [-1e300], #Z0 = 0
        [0], #H0 = 0,
        obs,
        [findfirst(iszero,obs[:,o])-1 for o in 1:size(obs)[2]],
        source_priors,
        mix_prior,
        bg_scores, #precalculated background score
        posterior_switch,
        Vector{String}(),
        no_models+1,
```

```
        IPM_likelihood(init_logPWM_sources(source_priors, source_length_limits), obs, [findfirst(iszero,obs[:,o])-1 for o in 1:size(obs)[2]], bg_scores, falses(size(obs)[2],length(source_priors))))

function assemble_IPMs(path::String, no_models::Integer, source_priors::AbstractVector{<:Union{<:AbstractVector{<:Dirichlet{<:AbstractFloa
mix_prior::Tuple{BitMatrix,<:AbstractFloat}, bg_scores::AbstractArray{<:AbstractFloat}, obs::AbstractArray{<:Integer}, source_length_limits::UnitRange{<:Integer})
        ensemble_records = Vector{Model_Record}()
        !isdir(path) && mkpath(path)

        @assert size(obs)[2]==size(bg_scores)[2]

        @showprogress 1 "Assembling IPM ensemble..." for model_no in 1:no_models
                model_path = string(path,'/',model_no)
                if !isfile(model_path)
                        model = ICA_PWM_Model(string(model_no), source_priors, mix_prior, bg_scores, obs, source_length_limits)
                        serialize(model_path, model) #save the model to the ensemble directory
                        push!(ensemble_records, Model_Record(model_path,model.log_Li))
                else #interrupted assembly pick up from where we left off
                        model = deserialize(model_path)
                        push!(ensemble_records, Model_Record(model_path,model.log_Li))
                end
        end

        return ensemble_records, minimum([record.log_Li for record in ensemble_records])
end

```

```
function distributed_IPM_assembly(worker_pool::Vector{Int64},
↪ path::String, no_models::Integer,
↪ source_priors::AbstractVector{<:Union{<:AbstractVector{<:Dirichlet{<:AbstractFloa
↪ mix_prior::Tuple{BitMatrix,<:AbstractFloat},
↪ bg_scores::AbstractArray{<:AbstractFloat},
↪ obs::AbstractArray{<:Integer},
↪ source_length_limits::UnitRange{<:Integer})
        ensemble_records = Vector{Model_Record}()
        !isdir(path) && mkpath(path)

        @assert size(obs)[2]==size(bg_scores)[2]

    model_chan=
     ↪ RemoteChannel(()→Channel{ICA_PWM_Model}(length(worker_pool)))
    job_chan = RemoteChannel(()→Channel{Union{Tuple,Nothing}}(1))
        put!(job_chan,(source_priors, mix_prior, bg_scores, obs,
         ↪ source_length_limits))

        sequence_workers(worker_pool, worker_assemble, job_chan,
         ↪ model_chan)

        assembly_progress=Progress(no_models, desc="Assembling IPM
         ↪ ensemble ... ")

        model_counter=check_assembly!(ensemble_records, path, no_models,
         ↪ assembly_progress)

        while model_counter ≤ no_models
                wait(model_chan)
                candidate=take!(model_chan)
                model = ICA_PWM_Model(string(model_counter),
                 ↪ candidate.origin, candidate.sources,
                 ↪ candidate.source_length_limits, candidate.mix_matrix,
                 ↪ candidate.log_Li)
                model_path=string(path,'/',model_counter)
                serialize(model_path,model)
                push!(ensemble_records,
                 ↪ Model_Record(model_path,model.log_Li))
                model_counter+=1
                next!(assembly_progress)
        end

        take!(job_chan),put!(job_chan,nothing)
```

```
119
120         return ensemble_records, minimum([record.log_Li for record in
            ↪  ensemble_records])
121 end
122                                 function
                                    ↪  check_assembly!(ensemble_records::AbstractVector{
                                    ↪  path::String, no_models::Integer,
                                    ↪  assembly_progress::Progress)
123                                         counter=1
124                                         while counter ≤ no_models
125                                                 model_path=string(path,'/',counter)
126                                                 if isfile(model_path)
127                                                         model=deserialize(model_path)
128                                                         push!(ensemble_records,
                                                            ↪  Model_Record(model_path,m
129                                                         counter+=1
130                                                         next!(assembly_progress)
131                                                 else
132                                                         return counter
133                                                 end
134                                         end
135                                         return counter
136                                 end
137
138                                 function
                                    ↪  worker_assemble(job_chan::RemoteChannel,
                                    ↪  models_chan::RemoteChannel,
                                    ↪  comms_chan::RemoteChannel)
139                                         put!(comms_chan,myid())
140                                         wait(job_chan)
141                                         params=fetch(job_chan)
142                                         while !(fetch(job_chan) ===
                                            ↪  nothing)
143                                                 model=ICA_PWM_Model(string(myid()),pa
144                                                 put!(models_chan,model)
145                                         end
146                                 end
147
148 function Base.show(io::IO, e::IPM_Ensemble; nsrc=0, progress=false)
149         livec=[model.log_Li for model in e.models]
150         maxLH=maximum(livec)
151         printstyled(io, "ICA PWM Model Ensemble @ $(e.path)\n",
            ↪  bold=true)
```

```
        msg = @sprintf "Contour: %3.6e MaxLH:%3.3e Max/Naive:%3.3e log Evidence:%3.6e" e.contour maxLH (maxLH-e.naive_lh) e.log_Zi[end]
        println(io, msg)
        hist=UnicodePlots.histogram(livec, title="Ensemble Likelihood Distribution")
        show(io, hist)
        println()
        progress && return(nrows(hist.graphics)+6)
end
```

15.3.7 /src/ensemble/ensemble_utilities.jl

```
function ensemble_history(e::IPM_Ensemble, bins=25)
    !e.sample_posterior && throw(ArgumentError("This ensemble has no posterior samples to show a history for!"))
    livec=vcat([model.log_Li for model in e.models],[model.log_Li for model in e.posterior_samples])
    show(histogram(livec, nbins=bins))
end

function e_backup(e::IPM_Ensemble, tuner::Permute_Tuner)
    serialize(string(e.path,'/',"ens"), e)
    serialize(string(e.path,'/',"tuner"), tuner)
end

function clean_ensemble_dir(e::IPM_Ensemble, model_pad::Integer; ignore_warn=false)
    !ignore_warn && e.sample_posterior && throw(ArgumentError("Ensemble is set to retain posterior samples and its directory should not be cleaned!"))
    for file in readdir(e.path)
        !(file in vcat([basename(model.path) for model in e.models],"ens",[string(number) for number in e.model_counter-length(e.models)-model_pad:e.model_counter-1])) && rm(e.path*'/'*file)
    end
end

function complete_evidence(e::IPM_Ensemble)
```

```
    return final_logZ = logaddexp(e.log_Zi[end], (logsumexp([model.log_Li for model in e.models]) +  e.log_Xi[length(e.log_Li)] - log(length(e.models))))
end

function reset_ensemble!(e::IPM_Ensemble)
    new_e=deepcopy(e)
    for i in 1:length(e.models)
        if string(i) in [basename(record.path) for record in e.models]
            new_e.models[i]=e.models[findfirst(isequal(string(i)), [basename(record.path) for record in e.models])]
        else
            new_e.models[i]=e.posterior_samples[findfirst(isequal(string(i)), [basename(record.path) for record in e.posterior_samples])]
        end
    end

    new_e.contour=minimum([record.log_Li for record in new_e.models])

    new_e.log_Li=[new_e.log_Li[1]]
    new_e.log_Xi=[new_e.log_Xi[1]]
    new_e.log_wi=[new_e.log_wi[1]]
    new_e.log_Liwi=[new_e.log_Liwi[1]]
    new_e.log_Zi=[new_e.log_Zi[1]]
    new_e.Hi=[new_e.Hi[1]]

    new_e.posterior_samples=Vector{Model_Record}()

    new_e.model_counter=length(new_e.models)+1

    clean_ensemble_dir(new_e, 0; ignore_warn=true)
    isfile(e.path*"/tuner") && rm(e.path*"/tuner")
    serialize(e.path*"/ens", new_e)

    return new_e
end

function move_ensemble!(e::IPM_Ensemble,path::String)
    !isdir(path) && mkdir(path)
    for file in readdir(e.path)
        mv(e.path*'/'*file,path*'/'*file)
```

```
    end

    for (n,model) in enumerate(e.models)
        e.models[n]=Model_Record(path*'/'*basename(model.path), model.log_Li)
    end
    if e.sample_posterior
        for (n,model) in enumerate(e.posterior_samples)
            e.posterior_samples[n]=Model_Record(path*'/'*basename(model.path), model.log_Li)
        end
    end

    rm(e.path)
    e.path=path
    serialize(e.path*"/ens",e)
    return e
end

function copy_ensemble!(e::IPM_Ensemble,path::String)
    new_e=deepcopy(e)
    !isdir(path) && mkdir(path)
    for file in readdir(e.path)
        cp(e.path*'/'*file,path*'/'*file, force=true)
    end

    for (n,model) in enumerate(e.models)
        new_e.models[n]=Model_Record(path*'/'*basename(model.path), model.log_Li)
    end
    if e.sample_posterior
        for (n,model) in enumerate(e.posterior_samples)
            new_e.posterior_samples[n]=Model_Record(path*'/'*basename(model.path), model.log_Li)
        end
    end

    new_e.path=path
    serialize(new_e.path*"/ens",e)
    return new_e
end
```

```julia

function rewind_ensemble(e::IPM_Ensemble,rewind_idx)
    !e.sample_posterior && throw(ArgumentError("An ensemble not retaining posterior samples cannot be rewound!"))
    rewind_idx ≥ length(e.log_Li) && throw(ArgumentError("rewind_idx must be less than the current iterate!"))

    n=length(e.models)
    max_model_no=length(e.log_Li)+length(e.models)-1
    rewind_model_no=rewind_idx+length(e.models)-1
    new_e = deepcopy(e)

    rm_models=[string(name) for name in rewind_model_no+1:max_model_no]

    filter!(model→!(basename(model.path) in rm_models),new_e.models)
    filter!(model→!(basename(model.path) in rm_models),new_e.posterior_samples)

    while length(new_e.models) < n
        push!(new_e.models,pop!(new_e.posterior_samples))
    end
    new_e.contour=new_e.log_Li[rewind_idx]
    new_e.log_Li=new_e.log_Li[1:rewind_idx]
    new_e.log_Xi=new_e.log_Xi[1:rewind_idx]
    new_e.log_wi=new_e.log_wi[1:rewind_idx]
    new_e.log_Liwi=new_e.log_Liwi[1:rewind_idx]
    new_e.log_Zi=new_e.log_Zi[1:rewind_idx]
    new_e.Hi=new_e.Hi[1:rewind_idx]

    new_e.model_counter=length(new_e.models)+rewind_idx

    return new_e
end

function show_models(e::IPM_Ensemble,idxs)
    liperm=sortperm([model.log_Li for model in e.models],rev=true)
    for idx in idxs
        m=deserialize(e.models[liperm[idx]].path)
        show(m)
    end
end

function get_model(e::IPM_Ensemble,no)
```

```
    return deserialize(e.path*'/'*string(no))
end
```

15.3.8 /src/nested_sampler/converge_ensemble.jl

```
function converge_ensemble!(e::IPM_Ensemble, instruction::Permute_Instruct; max_iterates=typemax(Int64), backup::Tuple{Bool,Integer}=(false,0), clean::Tuple{Bool,Integer,Integer}=(false,0,0), verbose::Bool=false, converge_criterion::String="standard", converge_factor::AbstractFloat=.001, progargs... )
    N = length(e.models); curr_it=length(e.log_Li)

    curr_it>1 && isfile(e.path*"/tuner") ? (tuner=deserialize(e.path*"/tuner")) : (tuner = Permute_Tuner(instruction)) #restore tuner from saved if any
    wk_mon = Worker_Monitor([1])
    meter = ProgressNS(e, wk_mon, tuner, 0.; start_it=curr_it, progargs... )

    converge_check = get_convfunc(converge_criterion)
    while !converge_check(e, converge_factor) && (curr_it ≤ max_iterates)
        warn,step_report = nested_step!(e, instruction) #step the ensemble
        warn == 1 && #"1" passed for warn code means no workers persist; all have hit the permute limit
                (@error "Failed to find new models, aborting at current iterate."; return e) #if there is a warning, iust return the ensemble and print info
        curr_it += 1

        tune_weights!(tuner, step_report)
        instruction = tuner.inst

        backup[1] && curr_it%backup[2] == 0 && e_backup(e,tuner) #every backup interval, serialise the ensemble and instruction
        clean[1] &&  !e.sample_posterior && curr_it%clean[2] == 0 && clean_ensemble_dir(e,clean[3]) #every clean interval, remove old discarded models

        update!(meter, converge_check(e,converge_factor,vals=true)... )
```

```
    end

    if converge_check(e,converge_factor)
        final_logZ = complete_evidence(e)
        @info "Job done, sampled to convergence. Final logZ $final_logZ"

        e_backup(e,tuner)
        clean[1] && !e.sample_posterior && clean_ensemble_dir(e,0) #final clean
        return final_logZ
    elseif curr_it==max_iterates
        @info "Job done, sampled to maximum iterate $max_iterates. Convergence criterion not obtained."

        e_backup(e,tuner)
        clean[1] && !e.sample_posterior && clean_ensemble_dir(e,0) #final clean
        return e.log_Zi[end]
    end
end

function converge_ensemble!(e::IPM_Ensemble, instruction::Permute_Instruct, wk_pool::Vector{Int64}; max_iterates=typemax(Int64), backup::Tuple{Bool,Integer}=(false,0), clean::Tuple{Bool,Integer,Integer}=(false,0,0), verbose::Bool=false, converge_criterion::String="standard", converge_factor::AbstractFloat=.001, progargs... )
    N = length(e.models)

    converge_check = get_convfunc(converge_criterion)
    model_chan= RemoteChannel(()->Channel{Tuple{Union{ICA_PWM_Model,String},Integer, AbstractVector{<:Tuple}}}(10*length(wk_pool))) #channel to take EM iterates off of
    job_chan = RemoteChannel(()->Channel{Tuple{<:AbstractVector{<:Model_Record}, Float64, Union{Permute_Instruct,String}}}(1))
    put!(job_chan,(e.models, e.contour, instruction))

    if !converge_check(e,converge_factor) #sequence workers only if not already converged
        @async sequence_workers(wk_pool, permute_IPM, e, job_chan, model_chan)
```

```
    end

    curr_it=length(e.log_Li)
    wk_mon=Worker_Monitor(wk_pool)
    curr_it>1 && isfile(e.path*"/tuner") ? (tuner=deserialize(e.path*"/tuner")) : (tuner = Permute_Tuner(instruction)) #restore tuner from saved if any
    meter = ProgressNS(e, wk_mon, tuner, 0.; start_it=curr_it, progargs... )

    while !converge_check(e, converge_factor) && (curr_it ≤ max_iterates)
        warn, step_report = nested_step!(e, model_chan, wk_mon) #step the ensemble
        warn == 1 && #"1" passed for warn code means no workers persist; all have hit the permute limit
                (@error "All workers failed to find new models, aborting at current iterate."; return e) #if there is a warning, iust return the ensemble and print info
        curr_it += 1
        tune_weights!(tuner, step_report)
        instruction = tuner.inst
        take!(job_chan); put!(job_chan,(e.models,e.contour,instruction))
        backup[1] && curr_it%backup[2] == 0 && e_backup(e,tuner) #every backup interval, serialise the ensemble and instruction
        clean[1] && !e.sample_posterior && curr_it%clean[2] == 0 && clean_ensemble_dir(e,clean[3]) #every clean interval, remove old discarded models

        update!(meter, converge_check(e,converge_factor,vals=true)... )
    end

    take!(job_chan); put!(job_chan, (e.models, e.contour, "stop")) #stop instruction terminates worker functions

    if converge_check(e,converge_factor)
        final_logZ = complete_evidence(e)
        @info "Job done, sampled to convergence. Final logZ $final_logZ"

        e_backup(e,tuner)
        clean[1] && !e.sample_posterior && clean_ensemble_dir(e,0) #final clean
        return final_logZ
```

```
    elseif curr_it==max_iterates
        @info "Job done, sampled to maximum iterate $max_iterates. Convergence criterion not obtained."

        e_backup(e,tuner)
        clean[1] && !e.sample_posterior && clean_ensemble_dir(e,0) #final clean
        return e.log_Zi[end]
    end
end

                function evidence_converge(e, evidence_fraction; vals=false)
                    val=lps(findmax([model.log_Li for model in e.models])[1],  e.log_Xi[end])
                    thresh=lps(log(evidence_fraction),e.log_Zi[end])
                    vals ? (return val, thresh) : (return val<thresh)
                end

                function compress_converge(e, compression_ratio; vals=false)
                    val=findmax([model.log_Li for model in e.models])[1]-e.contour
                    thresh=compression_ratio
                    vals ? (return val, thresh) : (return val<thresh)
                end

                function get_convfunc(criterion)
                    if criterion == "standard"
                        return evidence_converge
                    elseif criterion == "compression"
                        return compress_converge
                    else
                        throw(ArgumentError("Convergence criterion $criterion not supported! Try \"standard\" or \"compression\"."))
                    end
                end
```

15.3.9 /src/nested_sampler/nested_step.jl

```
#### IMPLEMENTATION OF JEFF SKILLINGS' NESTED SAMPLING ALGORITHM ####
function nested_step!(e::IPM_Ensemble, instruction::Permute_Instruct)
    N = length(e.models) #number of sample models/particles on the posterior surface
    i = length(e.log_Li) #iterate number, index for last values
    j = i+1 #index for newly pushed values

    e.contour, least_likely_idx = findmin([model.log_Li for model in e.models])
    Li_model = e.models[least_likely_idx]

    #SELECT NEW MODEL, SAVE TO ENSEMBLE DIRECTORY, CREATE RECORD AND PUSH TO ENSEMBLE
    model_selected=false; step_report=0
    while !model_selected
        candidate,step_report=permute_IPM(e, instruction)
        if !(candidate===nothing)
            if !(candidate.log_Li in [m.log_Li for m in e.models])
                model_selected=true

                new_model_record = Model_Record(string(e.path,'/',e.model_counter), candidate.log_Li);
                e.sample_posterior && push!(e.posterior_samples, Li_model)#if sampling posterior, push the model record to the ensemble's posterior samples vector
                deleteat!(e.models, least_likely_idx)
                push!(e.models, new_model_record);

                final_model=ICA_PWM_Model(string(e.model_counter), candidate.origin, candidate.sources, candidate.source_length_limits, candidate.mix_matrix, candidate.log_Li, candidate.permute_blacklist)
                serialize(new_model_record.path, final_model)

                e.model_counter +=1
            end
        else
            push!(e.models, Li_model)
            return 1, step_report
        end
```

```
    end

    #UPDATE ENSEMBLE QUANTITIES
    push!(e.log_Li, minimum([model.log_Li for model in e.models])) #log likelihood of the least likely model - the current ensemble ll contour at Xi
    push!(e.log_Xi, -i/N) #log Xi - crude estimate of the iterate's enclosed prior mass
    push!(e.log_wi, lps(e.log_Xi[i], -((j+1)/N)-log(2))) #log width of prior mass spanned by the last step-trapezoidal approx
    push!(e.log_Liwi, lps(e.log_Li[j],e.log_wi[j])) #log likelihood + log width = increment of evidence spanned by iterate
    push!(e.log_Zi, logaddexp(e.log_Zi[i],e.log_Liwi[j]))    #log evidence
    #information- dimensionless quantity
    push!(e.Hi, lps(
            (exp(lps(e.log_Liwi[j],-e.log_Zi[j])) * e.log_Li[j]), #term1
            (exp(lps(e.log_Zi[i],-e.log_Zi[j])) * lps(e.Hi[i],e.log_Zi[i])), #term2
            -e.log_Zi[j])) #term3

    return 0, step_report
end

function nested_step!(e::IPM_Ensemble, model_chan::RemoteChannel, wk_mon::Worker_Monitor)
    N = length(e.models)+1 #number of sample models/particles on the posterior surface- +1 as one has been removed in the distributed dispatch for converge_ensemble
    i = length(e.log_Li) #iterate number, index for last values
    j = i+1 #index for newly pushed values

    e.contour, least_likely_idx = findmin([model.log_Li for model in e.models])
    Li_model = e.models[least_likely_idx]

    #SELECT NEW MODEL, SAVE TO ENSEMBLE DIRECTORY, CREATE RECORD AND REPLACE LEAST LIKELY MODEL
    model_selected=false;wk=0;step_report=0
    while !model_selected
        @async wait(model_chan)
        candidate,wk,step_report = take!(model_chan)
        if !(candidate=="quit")
```

```
            if (candidate.log_Li > e.contour) && !(candidate.log_Li in [m.log_Li for m in e.models])
                model_selected=true

                new_model_record = Model_Record(string(e.path,'/',e.model_counter), candidate.log_Li);
                e.sample_posterior && push!(e.posterior_samples, Li_model)#if sampling posterior, push the model record to the ensemble's posterior samples vector
                deleteat!(e.models, least_likely_idx)
                push!(e.models, new_model_record);

                final_model=ICA_PWM_Model(string(e.model_counter), candidate.origin, candidate.sources, candidate.source_length_limits, candidate.mix_matrix, candidate.log_Li, candidate.permute_blacklist)
                serialize(new_model_record.path, final_model)

                e.model_counter +=1
            end
            update_worker_monitor!(wk_mon,wk,true)
        else
            update_worker_monitor!(wk_mon,wk,false)
            !any(wk_mon.persist) && (return 1,step_report)
        end
    end

    #UPDATE ENSEMBLE QUANTITIES
    push!(e.log_Li, minimum([model.log_Li for model in e.models])) #log likelihood of the least likely model - the current ensemble ll contour at Xi
    push!(e.log_Xi, -i/N) #log Xi - crude estimate of the iterate's enclosed prior mass
    push!(e.log_wi, lps(e.log_Xi[i], -((j+1)/N)-log(2))) #log width of prior mass spanned by the last step-trapezoidal approx
    push!(e.log_Liwi, lps(e.log_Li[j],e.log_wi[j])) #log likelihood + log width = increment of evidence spanned by iterate
    push!(e.log_Zi, logaddexp(e.log_Zi[i],e.log_Liwi[j]))    #log evidence

    #information- dimensionless quantity
    push!(e.Hi, lps(
```

```
            (exp(lps(e.log_Liwi[j],-e.log_Zi[j])) * e.log_Li[j]), #term1
            (exp(lps(e.log_Zi[i],-e.log_Zi[j])) *
             ↪  lps(e.Hi[i],e.log_Zi[i])), #term2
            -e.log_Zi[j])) #term3

    return 0, step_report
end
```

15.3.10 /src/permutation/Permute_Tuner.jl

```
mutable struct Permute_Tuner
    inst::Permute_Instruct
    velocities::Matrix{Float64}
    successes::BitMatrix #(memoryxfunc)
    tabular_display::DataFrame
    time_history::Vector{Float64}
    override::Bool
end

"""
    Permute_Tuner(instruction)

Generate a Permute_Tuner for the given instruction.
"""
function Permute_Tuner(instruction::Permute_Instruct)
    nfuncs=length(instruction.funcs)
    funcnames=Vector{String}()
    vels=ones(TUNING_MEMORY*instruction.func_limit,nfuncs)
    succs=trues(TUNING_MEMORY*instruction.func_limit,nfuncs)
    for (idx,func) in enumerate(instruction.funcs)
        length(instruction.args[idx])>0 ?
         ↪  (kwstr="(+$(length(instruction.args[idx]))kwa)") : (kwstr="")
        push!(funcnames, string(nameof(func),kwstr))
    end

    tabular_display=DataFrame("Function"⇒funcnames,
     ↪  "Succeed"⇒zeros(Int64,nfuncs), "Fail"⇒zeros(Int64,
     ↪  nfuncs),"Velocity"⇒ones(nfuncs), "Weights"⇒instruction.weights)

    instruction.override_time>0. ? (override=true) : (override=false)
    return
     ↪  Permute_Tuner(instruction,vels,succs,tabular_display,zeros(CONVERGENCE_MEMORY
```

```
end

"""
    tune_weights!(tuner, call_report)

Given a call_report from permute_IPM(), adjust tuner's Permute_Instruct weights for function success rate and likelihood surface velocity.
"""
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]
            tuner.successes[:,funcidx]=update_sucvec!(tuner.successes[:,funcidx],
            tuner.tabular_display[funcidx,"Succeed"]+=1
        else
            tuner.successes[:,funcidx]=update_sucvec!(tuner.successes[:,funcidx],
            tuner.tabular_display[funcidx,"Fail"]+=1
        end
    end
    tuner.override && mean(tuner.time_history)>tuner.inst.override_time ? (tuner.inst.weights=tuner.inst.override_weights; tuner.tabular_display[!,"Weights"]=tuner.inst.override_weights) :
        update_weights!(tuner)
end

function update_velocity!(velvec,time,distance)
    popfirst!(velvec) #remove first value
    vel = distance - log(time)
    push!(velvec,distance-log(time))
end

function update_sucvec!(sucvec, bool)
    popfirst!(sucvec)
    push!(sucvec, bool)
end

```

```
function update_weights!(t::Permute_Tuner)
    mvels=[mean(t.velocities[:,n]) for n in 1:length(t.inst.funcs)]
    t.tabular_display[!,"Velocity"]=copy(mvels)

    any(i->i<0,mvels) && (mvels.+=-minimum(mvels)+1.) #to calculate weights, scale negative values into >1.
    pvec=[mvels[n]*(sum(t.successes[:,n])/length(t.successes[:,n])) for n in 1:length(t.inst.funcs)]
    pvec.//=sum(pvec)
    (any(pvec.<t.inst.min_clmps) || any(pvec.>t.inst.max_clmps)) && clamp_pvec!(pvec,t.inst.min_clmps,t.inst.max_clmps)

    @assert isprobvec(pvec)
    t.inst.weights=pvec; t.tabular_display[!,"Weights"]=pvec
end

"""
    clamp_pvec(pvec, tuner)

Clamp the values of a probability vector between the minimums and maximums provided by a Permute_Tuner.
"""
            function clamp_pvec!(pvec, min_clmps, max_clmps)
                #logic- first accumulate on low values, then distribute excess from high values
                any(pvec.<min_clmps) ? (low_clamped=false) : (low_clamped=true)
                while !low_clamped
                    vals_to_accumulate=pvec.<min_clmps
                    vals_to_deplete=pvec.>min_clmps


                    depletion=sum(min_clmps[vals_to_accumulate]•-pvec[vals_to_acc

                    pvec[vals_to_accumulate]•=min_clmps[vals_to_accumulate]

                    pvec[vals_to_deplete]•-=depletion/sum(vals_to_deplete)

                    !any(pvec.<min_clmps)&&(low_clamped=true)
                end

                any(pvec.>max_clmps) ? (high_clamped=false) : (high_clamped=true)
```

```
                while !high_clamped
                    vals_to_deplete=pvec.>max_clmps
                    vals_to_accumulate=pvec.<max_clmps


                    depletion=sum(pvec[vals_to_deplete]•-max_clmps[vals_to_deplet
                    pvec[vals_to_deplete]•=max_clmps[vals_to_deplete]

                    pvec[vals_to_accumulate].+=depletion/sum(vals_to_accumulate)

                    !any(pvec.>max_clmps)&&(high_clamped=true)
                end
            end

function Base.show(io::IO, tuner::Permute_Tuner; progress=false)
    show(io, tuner.tabular_display, rowlabel=:I, summary=false)
    progress && return(size(tuner.tabular_display,1)+4)
end
```

15.3.11 /src/permutation/orthogonality_helper.jl

```
##ORTHOGONALITY HELPER
function consolidate_srcs(con_idxs::Dict{Integer,Vector{Integer}},
    m::ICA_PWM_Model, obs_array::AbstractMatrix{<:Integer},
    obs_lengths::AbstractVector{<:Integer},
    bg_scores::AbstractMatrix{<:AbstractFloat}, contour::AbstractFloat,
    models::AbstractVector{<:Model_Record};
    iterates::Integer=length(m.sources)*2, remote=false)
    new_log_Li=-Inf;  iterate = 1
    T,O = size(obs_array); T=T-1; S = length(m.sources)
    new_sources=deepcopy(m.sources); new_mix=deepcopy(m.mix_matrix)
    a, cache = IPM_likelihood(new_sources, obs_array, obs_lengths,
        bg_scores, new_mix, true, true)

    while new_log_Li ≤ contour && iterate ≤ iterates #until we produce
        a model more likely than the lh contour or exceed iterates
        new_sources=deepcopy(m.sources); new_mix=deepcopy(m.mix_matrix)
        clean=Vector{Bool}(trues(O))

        for host_src in filter(!in(vcat(values(con_idxs)...)),
            keys(con_idxs)) #copy mix information to the source to be
            consolidated on as host
```

```
            for cons_src in con_idxs[host_src]
                new_mix[:,host_src]•=[new_mix[o,host_src] || new_mix[o,cons_src] for o in 1:size(new_mix,1)]
            end
        end

        remote ? (merger_m = deserialize(rand(models).path)) : (merger_m = remotecall_fetch(deserialize, 1, rand(models).path)) #randomly select a model to merge
        used_m_srcs=Vector{Int64}()

        for src in unique(vcat(values(con_idxs)...)) #replace all non-host sources with sources from a merger model unlike the one being removed
            distvec=[pwm_distance(m.sources[src][1],m_src[1]) for m_src in merger_m.sources]
            m_src=findmax(distvec)[2]
            while m_src in used_m_srcs
                distvec[m_src]=0.; m_src=findmax(distvec)[2]
                length(used_m_srcs)==length(merger_m.sources) && break;break
            end

            clean[new_mix[:,src]]•=false #mark dirty any obs that start with the source
            new_sources[src]=merger_m.sources[m_src]
            new_mix[:,src]=merger_m.mix_matrix[:,m_src]
            clean[new_mix[:,src]]•=false #mark dirty any obs that end with the source
            push!(used_m_srcs, m_src)
        end

        if consolidate_check(new_sources)[1] #if the new sources pass the consolidate check
            new_log_Li, cache = IPM_likelihood(new_sources, obs_array, obs_lengths, bg_scores, new_mix, true, true, cache, clean) #assess likelihood
        end

        iterate += 1
    end

```

```
    return ICA_PWM_Model("candidate","consolidated $(m.origin)", new_sources, m.source_length_limits, new_mix, new_log_Li)
end

function consolidate_check(sources::AbstractVector{<:Tuple{<:AbstractMatrix{<:AbstractFloa thresh=CONSOLIDATE_THRESH, revcomp=REVCOMP)
    pass=true
    lengthδmat=[size(src1[1],1) - size(src2[1],1) for src1 in sources, src2 in sources]
    cons_idxs=Dict{Integer,Vector{Integer}}()
    for src1 in 1:length(sources), src2 in src1+1:length(sources)
        if lengthδmat[src1,src2]==0
            revcomp ? (info_condition = (pwm_distance(sources[src1][1],sources[src2][1]) < thresh || pwm_distance(sources[src1][1],revcomp_pwm(sources[src2][1])) < thresh)) : (info_condition = (pwm_distance(sources[src1][1],sources[src2][1]) < thresh))
            if info_condition
                if !in(src1,keys(cons_idxs))
                    cons_idxs[src1]=[src2]; pass=false
                else
                    push!(cons_idxs[src1], src2)
                end
            end
        end
    end

    return pass, cons_idxs
end

                function pwm_distance(pwm1,pwm2)
                    minwml=min(size(pwm1,1),size(pwm2,1))
                    return sum([euclidean(exp.(pwm1[pos,:]), exp.(pwm2[pos,:])) for pos in 1:minwml])/minwml
                end
```

15.3.12 /src/permutation/permute_control.jl

```
mutable struct Permute_Instruct
    funcs::AbstractVector{<:Function}
    weights::AbstractVector{<:AbstractFloat}
    args::AbstractVector{<:AbstractVector{<:Tuple{<:Symbol,<:Any}}}
    model_limit::Integer
    func_limit::Integer
    min_clmps::AbstractVector{<:AbstractFloat}
    max_clmps::AbstractVector{<:AbstractFloat}
    override_time::AbstractFloat
    override_weights::AbstractVector{<:AbstractFloat}
    Permute_Instruct(funcs,
                     weights,
                     model_limit,
                     func_limit;
                     min_clmps=fill(.01,length(funcs)),
                     max_clmps=fill(1.,length(funcs)),
                     override_time=0.,
                     override_weights=zeros(length(funcs)),
                     args=[Vector{Tuple{Symbol,Any}}() for i in 1:length(funcs)])=assert_permute_instruct(funcs,weights,args,
                     && new(funcs,weights,args,model_limit,func_limit,min_clmps,max_c
end

function assert_permute_instruct(funcs,weights,args,model_limit,func_limit,min_clmps, max_clmps, override_time, override_weights)

    !(length(funcs)==length(args)==length(weights)==length(min_clmps)==length(max
    && throw(ArgumentError("A valid Permute_Instruct must have as many tuning weights and argument vectors as functions!"))
    model_limit<1 && throw(ArgumentError("Permute_Instruct limit on models to permute must be positive Integer!"))
    func_limit<1 && throw(ArgumentError("Permute_Instruct limit on fuction calls per model permtued must be positive Integer!"))
    any(min_clmps.≥max_clmps) && throw(ArgumentError("Permute_Instruct max_clmps must all be greater than corresponding min_clmps!"))
    sum(min_clmps)>1. && throw(ArgumentError("Sum of minimum clamps must be <1.0 to maintain a valid probability vector!"))
    any(max_clmps.>1.) && throw(ArgumentError("Permute_Tuner maximum clamps cannot be >1.0!"))
```

```
    override_time<0. && throw(ArgumentError("Permute_Instruct override_time must be 0 (disabled) or positive float!"))
    override_time>0. && !isprobvec(override_weights) && throw(ArgumentError("Permute_Instruct override_weights must be valid probvec!"))
    return true
end


#permutation routine function-
#general logic: receive array of permutation parameters, until a model more likely than the least is found:
#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
#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

#four permutation modes: source (iterative random changes to sources until model lh>contour or iterate limit reached)
#                                                -(iterates, weight shift freq per source base, length change freq per source, weight_shift_dist (a ContinuousUnivariateDistribution)) for permute params
#                                                mix (iterative random changes to mix matrix as above)
#                                                -(iterates, unitrange of # of moves)
#                                                init (iteratively reinitialize sources from priors)
#                                                -(iterates) for init params
#                                                merge (iteratively copy a source + mix matrix row from another model in the ensemble until lh>contour or iterate                                                limit reached)
#                                                -(iterates) for merpge params


function permute_IPM(e::IPM_Ensemble, instruction::Permute_Instruct)
```

```
    call_report=Vector{Tuple{Int64,Float64,Float64}}()
        for model = 1:instruction.model_limit
                m_record = rand(e.models)
        m = deserialize(m_record.path)

        model_blacklist=copy(m.permute_blacklist)
        filteridxs, filtered_funcs, filtered_weights, filtered_args = filter_permutes(instruction, model_blacklist)

        for call in 1:instruction.func_limit
            start=time()
            funcidx=rand(filtered_weights)
            permute_func=filtered_funcs[funcidx]

            pos_args,kw_args=get_permfunc_args(permute_func,e,m,filtered_args[fun
            new_m=permute_func(pos_args ... ;kw_args ... )

            push!(call_report,(filteridxs[funcidx],time()-start,new_m.log_Li - e.contour))

            if length(new_m.permute_blacklist) > 0 && new_m.log_Li == -Inf #in this case the blacklisted function will not work on this model at all; it should be removed from the functions to use
                vcat(model_blacklist,new_m.permute_blacklist)
                filteridxs, filtered_funcs, filtered_weights, filtered_args = filter_permutes(instruction, model_blacklist)
            end

                        dupecheck(new_m,m) && new_m.log_Li > e.contour && return new_m, call_report
                end
        end
        return nothing, call_report
end

function permute_IPM(e::IPM_Ensemble, job_chan::RemoteChannel, models_chan::RemoteChannel, comms_chan::RemoteChannel) #ensemble.models is partially updated on the worker to populate arguments for permute funcs
        persist=true
    id=myid()
```

```
    put!(comms_chan,id)
    while persist
        wait(job_chan)
        start=time()
        e.models, e.contour, instruction = fetch(job_chan)
        instruction == "stop" && (persist=false; break)

        call_report=Vector{Tuple{Int64,Float64,Float64}}()
        for model=1:instruction.model_limit
                        found::Bool=false
            m_record = rand(e.models)

            remotecall_fetch(isfile,1,m_record.path) ? m = (remotecall_fetch(deserialize,1,m_record.path)) : (break)

            model_blacklist=copy(m.permute_blacklist)
            filteridxs, filtered_funcs, filtered_weights, filtered_args = filter_permutes(instruction, model_blacklist)

            for call in 1:instruction.func_limit
                start=time()
                funcidx=rand(filtered_weights)
                permute_func=filtered_funcs[funcidx]

                pos_args,kw_args=get_permfunc_args(permute_func,e,m,filtered_args
                new_m=permute_func(pos_args ... ;kw_args ... )

                push!(call_report,(filteridxs[funcidx],time()-start,new_m.log_Li - e.contour))

                if length(new_m.permute_blacklist) > 0 && new_m.log_Li == -Inf #in this case the blacklisted function will not work on this model at all; it should be removed from the functions to use
                    vcat(model_blacklist,new_m.permute_blacklist)
                    filteridxs, filtered_funcs, filtered_weights, filtered_args = filter_permutes(instruction, model_blacklist)
                end

                dupecheck(new_m,m) && new_m.log_Li > e.contour && ((put!(models_chan, (new_m ,id, call_report))); found=true; break)
```

```
                        end
            found==true && break;
            wait(job_chan)
            fetch(job_chan)≠e.models && (break) #if the ensemble has changed during the search, update it
                        model==instruction.model_limit && (put!(models_chan, ("quit", id, call_report));persist=false)#worker to put "quit" on channel if it fails to find a model more likely than contour
                end
        end
end
                function filter_permutes(instruction, model_blacklist)

                    filteridxs=findall(p→!in(p,model_blacklist),instruction.func
                    filtered_funcs=instruction.funcs[filteridxs]
                    filtered_weights=filter_weights(instruction.weights, filteridxs)
                    filtered_args=instruction.args[filteridxs]
                    return filteridxs, filtered_funcs, filtered_weights, filtered_args
                end

                function filter_weights(weights, idxs)
                    deplete=0.
                    for (i, weight) in enumerate(weights)
                        !in(i, idxs) && (deplete+=weight)
                    end
                    weights[idxs].+=deplete/length(idxs)
                    new_weights=weights[idxs]./sum(weights[idxs])
                    return Categorical(new_weights)
                end

                function get_permfunc_args(func::Function,e::IPM_Ensemble, m::ICA_PWM_Model, args::Vector{Tuple{Symbol,Any}})
                    pos_args=[]
                    argparts=Base.arg_decl_parts(methods(func).ms[1])
                    argnames=[Symbol(argparts[2][n][1]) for n in 2:length(argparts[2])]
```

```
        for argname in argnames #assemble basic positional arguments from ensemble and model fields
            if argname == Symbol('m')
                push!(pos_args,m)
            elseif argname in fieldnames(IPM_Ensemble)
                push!(pos_args,getfield(e,argname))
            elseif argname in fieldnames(ICA_PWM_Model)
                push!(pos_args,getfield(m,argname))
            else
                throw(ArgumentError("Positional argument $argname of $func not available in the ensemble or model!"))
            end
        end

        kw_args = NamedTuple()
        if length(args) > 0 #if there are any keyword arguments to pass
            sym=Vector{Symbol}()
            val=Vector{Any}()
            for arg in args
                push!(sym, arg[1]); push!(val, arg[2])
            end
            kw_args = (;zip(sym,val)...)
        end

        return pos_args, kw_args
    end

    function dupecheck(new_model, model)
        (new_model.sources==model.sources && new_model.mix_matrix==model.mix_matrix) ? (return false) : (return true)
    end
```

15.3.13 /src/permutation/permute_functions.jl

```
#DECORRELATION SEARCH PATTERNS


```

```
function permute_source(m::ICA_PWM_Model, models::Vector{Model_Record},
    obs_array::AbstractMatrix{<:Integer},
    obs_lengths::AbstractVector{<:Integer},
    bg_scores::AbstractMatrix{<:AbstractFloat}, contour::AbstractFloat,
    source_priors::AbstractVector{<:Union{<:AbstractVector{<:Dirichlet{<:AbstractFloa
    iterates::Integer=length(m.sources)*2,
    weight_shift_freq::AbstractFloat=PWM_SHIFT_FREQ,
    weight_shift_dist::Distributions.ContinuousUnivariateDistribution=PWM_SHIFT_DIST,
    length_change_freq::AbstractFloat=PWM_LENGTHPERM_FREQ,
    length_perm_range::UnitRange{<:Integer}=LENGTHPERM_RANGE,
    remote=false)
#weight_shift_dist is given in decimal probability values- converted to
    log space in permute_source_lengths!
    new_log_Li=-Inf;  iterate = 1
    O = size(obs_array,2);S = length(m.sources)
    new_sources=deepcopy(m.sources);

    a, cache = IPM_likelihood(m.sources, obs_array, obs_lengths,
        bg_scores, m.mix_matrix, true, true)

    while new_log_Li ≤ contour && iterate ≤ iterates #until we produce
        a model more likely than the lh contour or exceed iterates
        new_sources=deepcopy(m.sources);
        s = rand(1:S)
        clean=Vector{Bool}(trues(O))
        clean[m.mix_matrix[:,s]]•=false #all obs with source are dirty

        weight_shift_freq > 0 &&
            (new_sources[s]=permute_source_weights(new_sources[s],
            weight_shift_freq, weight_shift_dist))
        rand() < length_change_freq &&
            (new_sources[s]=permute_source_length(new_sources[s],
            source_priors[s], m.source_length_limits, length_perm_range))

        new_log_Li, cache = IPM_likelihood(new_sources, obs_array,
            obs_lengths, bg_scores, m.mix_matrix, true, true, cache,
            clean)
        iterate += 1
    end

    cons_check, cons_idxs = consolidate_check(new_sources)
```

```
    cons_check ? (return ICA_PWM_Model("candidate", "PS from $(m.name)", new_sources, m.source_length_limits, m.mix_matrix, new_log_Li)) : (return consolidate_srcs(cons_idxs, ICA_PWM_Model("candidate", "PS from $(m.name)", new_sources, m.source_length_limits, m.mix_matrix, new_log_Li), obs_array, obs_lengths, bg_scores, contour, models; remote=remote))
end

function permute_mix(m::ICA_PWM_Model, obs_array::AbstractMatrix{<:Integer}, obs_lengths::AbstractVector{<:Integer}, bg_scores::AbstractMatrix{<:AbstractFloat}, contour::AbstractFloat; iterates::Integer=10, mix_move_range::UnitRange=1:length(m.mix_matrix), remote=false)
    new_log_Li=-Inf;  iterate = 1; O = size(obs_array,2);
    new_mix=falses(size(m.mix_matrix))

    a, cache = IPM_likelihood(m.sources, obs_array, obs_lengths, bg_scores, m.mix_matrix, true, true)
    dirty=false

    while (new_log_Li ≤ contour || !dirty) && iterate ≤ iterates #until we produce a model more likely than the lh contour or exceed iterates
        mix_moves=rand(mix_move_range)
        mix_moves > length(m.mix_matrix) && (mix_moves = length(m.mix_matrix))

        new_mix, clean = mix_matrix_decorrelate(m.mix_matrix, mix_moves) #generate a decorrelated candidate mix
        c_log_li, c_cache = IPM_likelihood(m.sources, obs_array, obs_lengths, bg_scores, new_mix, true, true, cache, clean) #calculate the model with the candidate mix
        positive_indices=c_cache.>(cache) #obtain any obs indices that have greater probability than we started with
        clean=Vector{Bool}(trues(O)-positive_indices)
        if any(positive_indices) #if there are any such indices
            new_log_Li, cache = IPM_likelihood(m.sources, obs_array, obs_lengths, bg_scores, new_mix, true, true, cache, clean) #calculate the new model
            dirty=true
        end
        iterate += 1
```

```
    end

    return ICA_PWM_Model("candidate","PM from $(m.name)", m.sources, m.source_length_limits, new_mix, new_log_Li, Vector{Function}())  #no consolidate check is necessary as sources havent changed
end

function perm_src_fit_mix(m::ICA_PWM_Model, models::Vector{Model_Record}, obs_array::AbstractMatrix{<:Integer}, obs_lengths::AbstractVector{<:Integer}, bg_scores::AbstractMatrix{<:AbstractFloat},contour::AbstractFloat, source_priors::AbstractVector{<:Union{<:AbstractVector{<:Dirichlet{<:AbstractFloa iterates::Integer=length(m.sources)*2, weight_shift_freq::AbstractFloat=PWM_SHIFT_FREQ, length_change_freq::AbstractFloat=PWM_LENGTHPERM_FREQ, length_perm_range::UnitRange{<:Integer}=LENGTHPERM_RANGE,weight_shift_dist::Distr remote=false)
    new_log_Li=-Inf;  iterate = 1
    O = size(obs_array,2); S = length(m.sources)
    new_sources=deepcopy(m.sources); new_mix=deepcopy(m.mix_matrix)
    fit_mix in m.permute_blacklist ? (new_bl=m.permute_blacklist) : (new_bl=Vector{Function}())

    while new_log_Li ≤ contour && iterate ≤ iterates #until we produce a model more likely than the lh contour or exceed iterates
        new_sources=deepcopy(m.sources); new_mix=deepcopy(m.mix_matrix); tm_one=deepcopy(m.mix_matrix);
        clean=Vector{Bool}(trues(O))
        s = rand(1:S);
        clean[new_mix[:,s]]•=false #all obs starting with source are dirty

        new_mix[:,s]•=false;tm_one[:,s]•=true

        weight_shift_freq > 0 && (new_sources[s]=permute_source_weights(new_sources[s], weight_shift_freq, weight_shift_dist))
        rand() < length_change_freq && (new_sources[s]=permute_source_length(new_sources[s], source_priors[s], m.source_length_limits, length_perm_range))

        l,zero_cache=IPM_likelihood(new_sources, obs_array, obs_lengths, bg_scores, new_mix, true, true)
```

```
        l,one_cache=IPM_likelihood(new_sources, obs_array, obs_lengths, bg_scores, tm_one, true, true)
        fit_mix=one_cache.≥zero_cache

        if REVCOMP #if we're looking at reverse strands, we want to see if sources fit better on the reverse strand, and if so switch over to the revcomp source and use its fitted mix
            revsrc=(revcomp_pwm(new_sources[s][1]),new_sources[s][2])
            revsrcs=deepcopy(new_sources)
            revsrcs[s]=revsrc
            l, revsrc_cache=IPM_likelihood(revsrcs, obs_array, obs_lengths, bg_scores, tm_one, true, true)
            rfit_mix=revsrc_cache.≥zero_cache
            lps(revsrc_cache[rfit_mix])>lps(one_cache[fit_mix]) && (new_sources=revsrcs;fit_mix=rfit_mix) #if the total likelihood contribution for the revsource fit is greater than the source fit, take the revsource
        end

        new_mix[:,s]=fit_mix

        clean[fit_mix]•=false #all obs ending with source are dirty

        new_log_Li = IPM_likelihood(new_sources, obs_array, obs_lengths, bg_scores, new_mix, true, false, zero_cache, clean)
        iterate += 1
    end

    cons_check, cons_idxs = consolidate_check(new_sources)
    cons_check ? (return ICA_PWM_Model("candidate","PSFM from $(m.name)",new_sources, m.source_length_limits, new_mix, new_log_Li, new_bl)) : (return consolidate_srcs(cons_idxs, ICA_PWM_Model("candidate","PSFM from $(m.name)",new_sources, m.source_length_limits, new_mix, new_log_Li, new_bl), obs_array, obs_lengths, bg_scores, contour, models; remote=remote))
end

function fit_mix(m::ICA_PWM_Model, models::Vector{Model_Record}, obs_array::AbstractMatrix{<:Integer}, obs_lengths::AbstractVector{<:Integer}, bg_scores::AbstractMatrix{<:AbstractFloat}; remote=false)
    new_sources=deepcopy(m.sources); new_mix=deepcopy(m.mix_matrix); test_mix=falses(size(m.mix_matrix))
```

```
    new_bl=[fit_mix]

    l,zero_cache=IPM_likelihood(m.sources, obs_array, obs_lengths, bg_scores, test_mix, true, true)
    for (s,source) in enumerate(m.sources)
        test_mix=falses(size(m.mix_matrix))
        test_mix[:,s]•=true
        l,src_cache=IPM_likelihood(m.sources, obs_array, obs_lengths, bg_scores, test_mix, true, true)
        fit_mix=src_cache.≥zero_cache

        if REVCOMP #if we're looking at reverse strands, we want to see if sources fit better on the reverse strand, and if so switch over to the revcomp source and use its fitted mix
            revsrc=(revcomp_pwm(source[1]),source[2])
            revsrcs=deepcopy(new_sources)
            revsrcs[s]=revsrc
            l, revsrc_cache=IPM_likelihood(revsrcs, obs_array, obs_lengths, bg_scores, test_mix, true, true)
            rfit_mix=revsrc_cache.≥zero_cache
            lps(revsrc_cache[rfit_mix])>lps(src_cache[fit_mix]) && (new_sources=revsrcs;fit_mix=rfit_mix) #if the total likelihood contribution for the revsource fit is greater than the source fit, take the revsource
        end

        new_mix[:,s]=fit_mix
    end
    new_mix==m.mix_matrix ? (new_log_Li=-Inf) : (new_log_Li = IPM_likelihood(m.sources, obs_array, obs_lengths, bg_scores, new_mix, true)) #bail if the mix matrix doesnt change
    new_log_Li in [model.log_Li for model in models] && (new_log_Li=-Inf) #an existing identical log_Li is almost certainly a fitted model from another origin with the same sources, in this case abort to prevent dupes

    return ICA_PWM_Model("candidate","FM from $(m.name)",new_sources, m.source_length_limits, new_mix, new_log_Li, new_bl) #no consolidate check necessary as no change to sources
end

```

```
function random_decorrelate(m::ICA_PWM_Model, models::Vector{Model_Record}, obs_array::AbstractMatrix{<:Integer}, obs_lengths::AbstractVector{<:Integer}, bg_scores::AbstractMatrix{<:AbstractFloat}, contour::AbstractFloat, source_priors::AbstractVector{<:Union{<:AbstractVector{<:Dirichlet{<:AbstractFloa iterates::Integer=length(m.sources)*2, source_permute_freq::AbstractFloat=SRC_PERM_FREQ, weight_shift_freq::AbstractFloat=PWM_SHIFT_FREQ, length_change_freq::AbstractFloat=PWM_LENGTHPERM_FREQ, weight_shift_dist::Distributions.ContinuousUnivariateDistribution=PWM_SHIFT_DIST, mix_move_range::UnitRange=1:size(m.mix_matrix,1), remote=false)
    new_log_Li=-Inf;  iterate = 1
    O = size(obs_array,2); S = length(m.sources)
    new_sources=deepcopy(m.sources); new_mix=deepcopy(m.mix_matrix)

    a, cache = IPM_likelihood(new_sources, obs_array, obs_lengths, bg_scores, new_mix, true, true)

    while new_log_Li ≤ contour && iterate ≤ iterates #until we produce a model more likely than the lh contour or exceed iterates
        new_sources=deepcopy(m.sources); new_mix=deepcopy(m.mix_matrix)
        clean=Vector{Bool}(trues(O))
        s = rand(1:S)
        clean[new_mix[:,s]]•=false #all obs starting with source are dirty
        rand() < source_permute_freq && (new_sources[s]=permute_source_weights(new_sources[s], weight_shift_freq, weight_shift_dist))
        rand() < length_change_freq && (new_sources[s]=permute_source_length(new_sources[s], source_priors[s], m.source_length_limits))

        new_mix[:,s]=mixvec_decorrelate(new_mix[:,s],rand(mix_move_range))
        clean[new_mix[:,s]]•=false #all obs ending with source are dirty
        new_log_Li, cache = IPM_likelihood(new_sources, obs_array, obs_lengths, bg_scores, new_mix, true, true, cache, clean)
        iterate += 1
    end

    cons_check, cons_idxs = consolidate_check(new_sources)
```

```
    cons_check ? (return ICA_PWM_Model("candidate", "RD from $(m.name)", new_sources, m.source_length_limits, new_mix, new_log_Li)) : (return consolidate_srcs(cons_idxs, ICA_PWM_Model("candidate","RD from $(m.name)",new_sources, m.source_length_limits, new_mix, new_log_Li), obs_array, obs_lengths, bg_scores, contour, models; remote=remote))
end

function shuffle_sources(m::ICA_PWM_Model, models::AbstractVector{<:Model_Record}, obs_array::AbstractMatrix{<:Integer}, obs_lengths::AbstractVector{<:Integer}, bg_scores::AbstractMatrix{<:AbstractFloat}, contour::AbstractFloat; remote=false)
    new_log_Li=-Inf; O = size(obs_array,2); S = length(m.sources)
    new_sources=deepcopy(m.sources); new_mix=deepcopy(m.mix_matrix)

    a, cache = IPM_likelihood(m.sources, obs_array, obs_lengths, bg_scores, m.mix_matrix, true, true)

    remote ? (merger_m = deserialize(rand(models).path)) : (merger_m = remotecall_fetch(deserialize, 1, rand(models).path))#randomly select a model to merge

    svec=[1:S...]

    while new_log_Li ≤ contour && length(svec)>0 #until we produce a model more likely than the lh contour or no more sources to attempt merger
        new_sources=deepcopy(m.sources); new_mix=deepcopy(m.mix_matrix)
        clean=Vector{Bool}(trues(O))

        s = popat!(svec,rand(1:length(svec))) #randomly select a source to merge

        new_sources[s]=merger_m.sources[s]; new_mix[:,s]•=merger_m.mix_matrix[:,s] #shuffle in the merger model source from this index

        clean[m.mix_matrix[:,s]]•=false #mark dirty any obs that have the source
```

```
        new_log_Li, cache = IPM_likelihood(new_sources, obs_array, obs_lengths, bg_scores, m.mix_matrix, true, true, cache, clean) #assess likelihood
    end

    cons_check, cons_idxs = consolidate_check(new_sources)
    cons_check ? (return ICA_PWM_Model("candidate","SS from $(m.name)",new_sources, m.source_length_limits, new_mix, new_log_Li,Vector{Function}())) : (return consolidate_srcs(cons_idxs, ICA_PWM_Model("candidate","SS from $(m.name)",new_sources, m.source_length_limits, new_mix, new_log_Li), obs_array, obs_lengths, bg_scores, contour, models; remote=remote))
end

function accumulate_mix(m::ICA_PWM_Model, models::AbstractVector{<:Model_Record}, obs_array::AbstractMatrix{<:Integer}, obs_lengths::AbstractVector{<:Integer}, bg_scores::AbstractMatrix{<:AbstractFloat}, contour::AbstractFloat; remote=false)
    new_log_Li=-Inf; O = size(obs_array,2); S = length(m.sources)
    new_sources=deepcopy(m.sources); new_mix=deepcopy(m.mix_matrix)
    new_bl=Vector{Function}()

    a, cache = IPM_likelihood(m.sources, obs_array, obs_lengths, bg_scores, m.mix_matrix, true, true)

    remote ? (merger_m = deserialize(rand(models).path)) : (merger_m = remotecall_fetch(deserialize, 1, rand(models).path))#randomly select a model to merge

    svec=[1:S ... ]

    while new_log_Li ≤ contour && length(svec)>0 #until we produce a model more likely than the lh contour or no more sources to attempt merger
        new_sources=deepcopy(m.sources); new_mix=deepcopy(m.mix_matrix)
        clean=Vector{Bool}(trues(O))

        s = popat!(svec,rand(1:length(svec))) #randomly select a source to merge
```

```
        distvec=[pwm_distance(src[1],m_src[1]) for src in m.sources, m_src in merger_m.sources]
        S > 1 ? merge_s=findmin(distvec)[2][2] : merge_s=findmin(distvec)[2]

        new_mix[:,s]•=[new_mix[o,s] || merger_m.mix_matrix[o,merge_s] for o in 1:size(new_mix,1)] #accumulate the mix vector for this source

        clean[m.mix_matrix[:,s]]•=false #mark dirty any obs that have the source
        new_log_Li, cache = IPM_likelihood(new_sources, obs_array, obs_lengths, bg_scores, m.mix_matrix, true, true, cache, clean) #assess likelihood

        if new_log_Li ≤ contour #if accumulating on the host source doesnt work, try copying over the merger source
            new_sources[s]=merger_m.sources[merge_s]
            new_log_Li, cache = IPM_likelihood(new_sources, obs_array, obs_lengths, bg_scores, m.mix_matrix, true, true, cache, clean) #assess likelihood
        end
    end

    (new_log_Li == m.log_Li || new_log_Li in [model.log_Li for model in models]) && (new_log_Li=-Inf) #accumulate can sometimes duplicate models if source mix is identical

    cons_check, cons_idxs = consolidate_check(new_sources)
    cons_check ? (return ICA_PWM_Model("candidate","AM from $(m.name)",new_sources, m.source_length_limits, new_mix, new_log_Li)) : (return consolidate_srcs(cons_idxs, ICA_PWM_Model("candidate","AM from $(m.name)",new_sources, m.source_length_limits, new_mix, new_log_Li), obs_array, obs_lengths, bg_scores, contour, models; remote=remote))
end

function distance_merge(m::ICA_PWM_Model, models::AbstractVector{<:Model_Record}, obs_array::AbstractMatrix{<:Integer}, obs_lengths::AbstractVector{<:Integer}, bg_scores::AbstractMatrix{<:AbstractFloat}, contour::AbstractFloat; remote=false)
```

```
    new_log_Li=-Inf; O = size(obs_array,2); S = length(m.sources)
    new_sources=deepcopy(m.sources); new_mix=deepcopy(m.mix_matrix)

    a, cache = IPM_likelihood(new_sources, obs_array, obs_lengths, bg_scores, new_mix, true, true)

    remote ? (merger_m = deserialize(rand(models).path)) : (merger_m = remotecall_fetch(deserialize, 1, rand(models).path)) #randomly select a model to merge

    svec=[1:S ... ]

    while new_log_Li ≤ contour && length(svec)>0 #until we produce a model more likely than the lh contour or no more sources to attempt merger
        new_sources=deepcopy(m.sources); new_mix=deepcopy(m.mix_matrix)
        clean=Vector{Bool}(trues(O))

        s = popat!(svec,rand(1:length(svec))) #randomly select a source to merge
        merge_s=most_dissimilar(new_mix[:,s],merger_m.mix_matrix) #find the source in the merger model whose mixvec is most dissimilar to the one selected

        clean[new_mix[:,s]]•=false #mark dirty any obs that start with the source
        new_sources[s] = merger_m.sources[merge_s] #copy the source
        new_mix[:,s] = merger_m.mix_matrix[:,merge_s] #copy the mixvector (without which the source will likely be highly improbable)
        clean[new_mix[:,s]]•=false #mark dirty any obs that end with the source

        new_log_Li, cache = IPM_likelihood(new_sources, obs_array, obs_lengths, bg_scores, new_mix, true, true, cache, clean) #assess likelihood
    end

    cons_check, cons_idxs = consolidate_check(new_sources)
```

```
    cons_check ? (return ICA_PWM_Model("candidate","DM from $(m.name)",new_sources, m.source_length_limits, new_mix, new_log_Li,Vector{Function}())) : (return consolidate_srcs(cons_idxs, ICA_PWM_Model("candidate","DM from $(m.name)",new_sources, m.source_length_limits, new_mix, new_log_Li), obs_array, obs_lengths, bg_scores, contour, models; remote=remote))
end

function similarity_merge(m::ICA_PWM_Model, models::AbstractVector{<:Model_Record}, obs_array::AbstractMatrix{<:Integer}, obs_lengths::AbstractVector{<:Integer}, bg_scores::AbstractMatrix{<:AbstractFloat}, contour::AbstractFloat; remote=false)
    new_log_Li=-Inf; O = size(obs_array,2); S = length(m.sources)
    new_sources=deepcopy(m.sources)

    a, cache = IPM_likelihood(m.sources, obs_array, obs_lengths, bg_scores, m.mix_matrix, true, true)

    remote ? (merger_m = deserialize(rand(models).path)) : (merger_m = remotecall_fetch(deserialize, 1, rand(models).path))#randomly select a model to merge

    svec=[1:S ... ]

    while new_log_Li ≤ contour && length(svec)>0 #until we produce a model more likely than the lh contour or no more sources to attempt merger
        new_sources=deepcopy(m.sources); new_mix=deepcopy(m.mix_matrix)
        clean=Vector{Bool}(trues(O))

        s = popat!(svec,rand(1:length(svec))) #randomly select a source to merge
        merge_s=most_similar(m.mix_matrix[:,s],merger_m.mix_matrix) #obtain the source in the merger model whose mixvec is most similar to the one in the original

        clean[m.mix_matrix[:,s]]•=false #mark dirty any obs that have the source
        new_sources[s] = merger_m.sources[merge_s] #copy the source, but dont copy the mixvector
```

```
        new_log_Li, cache = IPM_likelihood(new_sources, obs_array, obs_lengths, bg_scores, m.mix_matrix, true, true, cache, clean) #assess likelihood
    end

    cons_check, cons_idxs = consolidate_check(new_sources)
    cons_check ? (return ICA_PWM_Model("candidate","SM from $(m.name)",new_sources, m.source_length_limits, m.mix_matrix, new_log_Li,Vector{Function}())) : (return consolidate_srcs(cons_idxs, ICA_PWM_Model("candidate","SM from $(m.name)",new_sources, m.source_length_limits, m.mix_matrix, new_log_Li), obs_array, obs_lengths, bg_scores, contour, models; remote=remote))
end

function reinit_src(m::ICA_PWM_Model, models::AbstractVector{<:Model_Record}, obs_array::AbstractMatrix{<:Integer}, obs_lengths::AbstractVector{<:Integer}, bg_scores::AbstractMatrix{<:AbstractFloat},contour::AbstractFloat, source_priors::AbstractVector{<:Union{<:AbstractVector{<:Dirichlet{<:AbstractFloa
iterates::Integer=length(m.sources)*2, remote=false)
    new_log_Li=-Inf;  iterate = 1
    O = size(obs_array,2); S = length(m.sources)
    new_sources=deepcopy(m.sources); new_mix=deepcopy(m.mix_matrix)

    while new_log_Li ≤ contour && iterate ≤ iterates #until we produce a model more likely than the lh contour or exceed iterates
        new_sources=deepcopy(m.sources); new_mix=deepcopy(m.mix_matrix); tm_one=deepcopy(m.mix_matrix);
        clean=Vector{Bool}(trues(O))
        s = rand(1:S);
        clean[new_mix[:,s]]•=false #all obs starting with source are dirty

        new_mix[:,s]•=false;tm_one[:,s]•=true

        new_sources[s] = init_logPWM_sources([source_priors[s]], m.source_length_limits)[1] #reinitialize the source

        l,zero_cache=IPM_likelihood(new_sources, obs_array, obs_lengths, bg_scores, new_mix, true, true)
```

```
        l,one_cache=IPM_likelihood(new_sources, obs_array, obs_lengths, bg_scores, tm_one, true, true)

        fit_mix=one_cache.≥zero_cache

        new_mix[:,s]=fit_mix

        clean[fit_mix]•=false #all obs ending with source are dirty

        new_log_Li = IPM_likelihood(new_sources, obs_array, obs_lengths, bg_scores, new_mix, true, false, zero_cache, clean)
        iterate += 1
    end

    cons_check, cons_idxs = consolidate_check(new_sources)
    cons_check ? (return ICA_PWM_Model("candidate","RS from $(m.name)", new_sources, m.source_length_limits, new_mix, new_log_Li,Vector{Function}())) : (return consolidate_srcs(cons_idxs, ICA_PWM_Model("candidate","RS from $(m.name)",new_sources, m.source_length_limits, new_mix, new_log_Li), obs_array, obs_lengths, bg_scores, contour, models; remote=remote))
end

function erode_model(m::ICA_PWM_Model, models::AbstractVector{<:Model_Record}, obs_array::AbstractMatrix{<:Integer}, obs_lengths::AbstractVector{<:Integer}, bg_scores::AbstractMatrix{<:AbstractFloat}, contour::AbstractFloat; info_thresh::AbstractFloat=EROSION_INFO_THRESH, remote=false)
    new_log_Li=-Inf; O = size(obs_array,2);
    new_sources=deepcopy(m.sources)

    erosion_sources=Vector{Integer}()
    for (s,src) in enumerate(m.sources)
        pwm,pi=src
        if size(pwm,1)>m.source_length_limits[1]  #do not consider eroding srcs at min length limit
            infovec=get_pwm_info(pwm)
            any(info→<(info, info_thresh),infovec) && push!(erosion_sources,s)
        end
    end
```

```julia

    length(erosion_sources)==0 && return ICA_PWM_Model("candidate","EM from $(m.name)",new_sources, m.source_length_limits, m.mix_matrix, new_log_Li,[erode_model])#if we got a model we cant erode bail out with a model marked -Inf lh and with EM blacklisted

    a, cache = IPM_likelihood(m.sources, obs_array, obs_lengths, bg_scores, m.mix_matrix, true, true)

    while new_log_Li ≤ contour && length(erosion_sources) > 0 #until we produce a model more likely than the lh contour or there are no more sources to erode
        clean=Vector{Bool}(trues(O))
        s=popat!(erosion_sources,rand(1:length(erosion_sources)))

        new_sources[s]=erode_source(new_sources[s], m.source_length_limits, info_thresh)
        clean[m.mix_matrix[:,s]]•=false

        new_log_Li, cache = IPM_likelihood(new_sources, obs_array, obs_lengths, bg_scores, m.mix_matrix, true, true, cache, clean) #assess likelihood
    end

    new_log_Li ≤ contour ? (blacklist=[erode_model]) : (blacklist=Vector{Function}())

    cons_check, cons_idxs = consolidate_check(new_sources)
    cons_check ? (return ICA_PWM_Model("candidate","EM 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","EM 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))
end

```

```
function info_fill(m::ICA_PWM_Model, models::AbstractVector{<:Model_Record}, obs_array::AbstractMatrix{<:Integer}, obs_lengths::AbstractVector{<:Integer}, bg_scores::AbstractMatrix{<:AbstractFloat}, contour::AbstractFloat; remote=false)
    new_log_Li=-Inf; O = size(obs_array,2); S = length(m.sources); iterate=1
    new_sources=deepcopy(m.sources)

    a, cache = IPM_likelihood(m.sources, obs_array, obs_lengths, bg_scores, m.mix_matrix, true, true)

    svec=[1:S ... ]

    while new_log_Li ≤ contour && length(svec)>0 #until we produce a model more likely than the lh contour or no more sources to attempt infofill
        new_sources=deepcopy(m.sources)
        clean=Vector{Bool}(trues(O))
        s = popat!(svec,rand(1:length(svec))) #randomly select a source to merge
        pwm=new_sources[s][1]
        fill_idx=findmin(get_pwm_info(pwm))[2]
        fill_bases=[1,2,3,4]
        fill_candidate=findmax(pwm[fill_idx,:])[2]
        deleteat!(fill_bases, findfirst(b→b==fill_candidate,fill_bases))
        new_sources[s][1][fill_idx,:]⋅=-Inf; new_sources[s][1][fill_idx,fill_candidate]=0.
        clean[m.mix_matrix[:,s]]⋅=false
        new_log_Li, cache = IPM_likelihood(new_sources, obs_array, obs_lengths, bg_scores, m.mix_matrix, true, true, cache, clean) #assess likelihood

        while new_log_Li ≤contour && length(fill_bases)>0
            fill_candidate=popat!(fill_bases,rand(1:length(fill_bases)))
            new_sources[s][1][fill_idx,:]⋅=-Inf; new_sources[s][1][fill_idx,fill_candidate]=0.
            new_log_Li, cache = IPM_likelihood(new_sources, obs_array, obs_lengths, bg_scores, m.mix_matrix, true, true, cache, clean) #assess likelihood
        end
    end
```

```julia

    new_log_Li ≤ contour ? (blacklist=[info_fill]) : (blacklist=Vector{Function}())

    cons_check, cons_idxs = consolidate_check(new_sources)
    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))
end

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/permutation/permute_utilities.jl

```julia
##BASIC UTILITY FUNCTIONS
#SOURCE PERMUTATION
function permute_source_weights(source::Tuple{<:AbstractMatrix{<:AbstractFloat},<:Integer}, shift_freq::AbstractFloat, PWM_shift_dist::Distribution{Univariate,Continuous})
    dirty=false; source_length=size(source[1],1)
    new_source=deepcopy(source)

    for source_pos in 1:source_length
        if rand() ≤ shift_freq
            pos_WM = exp.(source[1][source_pos,:]) #leaving logspace, get the wm at that position
            new_source[1][source_pos,:] = log.(wm_shift(pos_WM, PWM_shift_dist)) #accumulate probabilty at a randomly selected base, reassign in logspace and carry on
            !dirty && (dirty=true)
        end
    end

    if !dirty #if no positions were shifted, pick one and shift
        rand_pos=rand(1:source_length)
```

```
        pos_WM = exp.(source[1][rand_pos,:])
        new_source[1][rand_pos,:]=log.(wm_shift(pos_WM, PWM_shift_dist))
    end

    return new_source
end

                function wm_shift(pos_WM::AbstractVector{<:AbstractFloat}, PWM_shift_dist::Distribution{Univariate,Continuous})
                    base_to_shift = rand(1:4) #pick a base to accumulate probability
                    permute_sign = rand(-1:2:1)
                    shift_size = rand(PWM_shift_dist)
                    new_wm=zeros(4)

                    for base in 1:4 #ACGT
                        if base == base_to_shift
                            new_wm[base] =
                            clamp(0, #no lower than 0 prob
                            (pos_WM[base]          #selected PWM posn
                            + permute_sign * shift_size), #randomly permuted by size param
                            1) #no higher than prob 1
                        else
                            size_frac = shift_size / 3 #other bases shifted in the opposite direction by 1/3 the shift accumulated at the base to permute
                            new_wm[base] =
                            clamp(0,
                            (pos_WM[base]
                            - permute_sign * size_frac),
                            1)
                        end
                    end
                    new_wm = new_wm ./ sum(new_wm) #renormalise to sum 1 - necessary in case of clamping at 0 or 1
                    !isprobvec(new_wm) && throw(DomainError(new_wm, "Bad weight vector generated in wm_shift!")) #throw assertion exception if the position WM is invalid
                    return new_wm
                end
```

```
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)

    permute_sign, permute_length = get_length_params(source_length, length_limits, permute_range)

    permute_sign==1 ? permute_pos = rand(1:source_length+1) :
        permute_pos=rand(1:source_length-permute_length)

    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
        upstream_source=source_PWM[1:permute_pos-1,:]
        downstream_source=source_PWM[permute_pos:end,:]
        source_PWM=vcat(upstream_source,ins_WM,downstream_source)
        permute_pos==1 && (prior_idx-=permute_length)
    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)
```

```
    end

    return (source_PWM, prior_idx) #return a new source
end

                function get_length_params(source_length::Integer, length_limits::UnitRange{<:Integer}, permute_range::UnitRange{<:Integer})
                    extendable = length_limits[end]-source_length
                    contractable =  source_length-length_limits[1]

                    if extendable == 0 && contractable > 0
                        permute_sign=-1
                    elseif contractable == 0 && extendable > 0
                        permute_sign=1
                    else
                        permute_sign = rand(-1:2:1)
                    end

                    permute_sign==1 && extendable<permute_range[end] && (permute_range=permute_range[1]:extendable)
                    permute_sign==-1 && contractable<permute_range[end] && (permute_range=permute_range[1]:contractable)
                    permute_length = rand(permute_range)

                    return permute_sign, permute_length
                end

function erode_source(source::Tuple{<:AbstractMatrix{<:AbstractFloat},<:Integer},length_li
    pwm,prior_idx=source
    infovec=get_pwm_info(pwm)
    start_idx,end_idx=get_erosion_idxs(infovec, info_thresh, length_limits)

    return new_source=(pwm[start_idx:end_idx,:], prior_idx+start_idx-1)
end

    function get_pwm_info(pwm::AbstractMatrix{<:AbstractFloat}; logsw::Bool=true)
        wml=size(pwm,1)
        infovec=zeros(wml)
        for pos in 1:wml
```

```
            logsw ? wvec=deepcopy(exp.(pwm[pos,:])) : wvec=deepcopy(pwm[pos,:])
            !isprobvec(wvec) && throw(DomainError(wvec, "Bad wvec in get_pwm_info -Original sources must be in logspace!!"))
            wvec.+=10^-99
            infscore = (2.0 + sum([x*log(2,x) for x in wvec]))
            infovec[pos]=infscore
        end
        return infovec
    end

    function get_erosion_idxs(infovec::AbstractVector{<:AbstractFloat}, info_thresh::AbstractFloat, length_limits::UnitRange{<:Integer})
        srcl=length(infovec)
        contractable =  srcl-length_limits[1]
        contractable ≤0 && throw(DomainError(contractable, "erode_source passed a source at its lower length limit!"))
        centeridx=findmax(infovec)[2]

        start_idx=findprev(info→<(info,info_thresh),infovec,centeridx)
        start_idx===nothing ? (start_idx=1) : (start_idx+=1)
        end_idx=findnext(info→<(info, info_thresh),infovec,centeridx)
        end_idx===nothing ? (end_idx=srcl) : (end_idx-=1)

        pos_to_erode=srcl-(end_idx-start_idx)
        if pos_to_erode > contractable
            pos_to_restore = pos_to_erode-contractable
            while pos_to_restore>0
                end_die=rand()
                if end_die ≤ .5
                    start_idx>1 && (pos_to_restore-=1; start_idx-=1)
                else
                    end_idx<srcl && (pos_to_restore-=1; end_idx+=1)
                end
            end
        end

        return start_idx, end_idx
    end

#MIX MATRIX FUNCTIONS
function mixvec_decorrelate(mix::BitVector, moves::Integer)
    new_mix=deepcopy(mix)
```

```
    idxs_to_flip=rand(1:length(mix), moves)
    new_mix[idxs_to_flip] ⊻= .!mix[idxs_to_flip]
    return new_mix
end

function mix_matrix_decorrelate(mix::BitMatrix, moves::Integer)
    clean=Vector{Bool}(trues(size(mix,1)))
    new_mix=deepcopy(mix)
    indices_to_flip = rand(CartesianIndices(mix), moves)
    new_mix[indices_to_flip] ⊻= .!mix[indices_to_flip]
    clean[unique([idx[1] for idx in indices_to_flip])] ⊻= false #mark all obs that had flipped indices dirty
    return new_mix, clean
end


# function most_dissimilar(mix1, mix2)
#     S1=size(mix1,2);S2=size(mix2,2)
#     dist_mat=zeros(S1,S2)
#     for s1 in 1:S1, s2 in 1:S2
#         dist_mat[s1,s2]=sum(mix1[:,s1].==mix2[:,s2])
#     end
#     scores=vec(sum(dist_mat,dims=1))
#     return findmin(scores)[2]
# end


function most_dissimilar(src_mixvec, target_mixmat)
    src_sim = [sum(src_mixvec.==target_mixmat[:,s]) for s in 1:size(target_mixmat,2)] #compose array of elementwise equality comparisons between mixvectors and sum to score
    merge_s=findmin(src_sim)[2] #source from merger model will be the one with the highest equality comparison score
end

function most_similar(src_mixvec, target_mixmat)
    src_sim = [sum(src_mixvec.==target_mixmat[:,s]) for s in 1:size(target_mixmat,2)] #compose array of elementwise equality comparisons between mixvectors and sum to score
    merge_s=findmax(src_sim)[2] #source from merger model will be the one with the highest equality comparison score
end
```

15.3.15 /src/utilities/model_display.jl

```
#THIS CODE BASED ON N. REDDY'S THICWEED.JL DISPLAY SCRIPT

#CONSTANTS
# each tuple specifies the x, y, fontsize, xscale to print the character
# in a 100×100 box at position 100,100.
charvals_dict = Dict{Char,Tuple}('A'⇒ (88.5,199.,135.,1.09),
                 'C'⇒ (79.,196.,129.,1.19),
                 'G'⇒ (83.,196.,129.,1.14),
                 'T'⇒ (80.,199.,135.,1.24))

colour_dict = Dict{Char,String}('A'⇒"(0,128,0)", #green
                 'C'⇒"(0,0,128)", #blue
                 'G'⇒"(255,69,0)", #yellow-brown
                 'T'⇒"(150,0,0)")  #red

#char output params for string display of PWMs in nested sampler instrumentation
lwcs_vec=['a','c','g','t']
upcs_vec=['A','C','G','T']
cs_vec=[:green,:blue,:yellow,:red]
thresh_dict=Dict([(0.,'_'),(.25,'.'),(.5,"lwcs"),(1,"upcs")])

source_left_x = 10
xlen = 20
yscale = 250
scale_factor = 0.9
ypixels_per_source = 250
ypad = 10
xpad = 20
xpixels_per_position = 40
fontsize1=60
fontsize2=40

#function to convert ICA_PWM_Model sources lists to PWM sequence logo diagrams
function logo_from_model(model::ICA_PWM_Model,svg_output::String;freq_sort::Bool=false)
    source_tups = Vector{Tuple{AbstractFloat, Integer, Matrix{AbstractFloat}}}() #(%of sequences w/ source, prior index, weight matrix)
    mix = model.mix_matrix #o x s
```

```
    for (prior, source) in enumerate(model.sources)
        push!(source_tups, (sum(model.mix_matrix[:,prior])/size(model.mix_matrix)[1], prior, exp.(source[1])))
    end

    freq_sort && sort(source_tups)

    file = open(svg_output, "w")
    write(file, svg_header(xpad+xpixels_per_position*maximum([length(source[1]) for source in model.sources]),ypixels_per_source*length(model.sources)+ypad))

    curry = 0
    for (frequency, index, source) in source_tups
        curry += yscale
        font1y = curry-190
        ndig = ndigits(index+1)
        write(file, pwm_to_logo(source,source_left_x,curry,xlen,yscale*scale_factor))
        write(file, "<text x=\"10\" y=\"$font1y\" font-family=\"Helvetica\" font-size=\"$fontsize1\" font-weight=\"bold\" >$(index)</text>")
        write(file, "<text x=\"$(20+fontsize1*0.6*ndig)\" y=\"$(font1y-fontsize1+1.5*fontsize2)\" font-family=\"Helvetica\" font-size=\"$fontsize2\" font-weight=\"bold\" >$(frequency*100) % of sequences</text>\n")
    end

    write(file, svg_footer())
    @info "Logo written."
    close(file)
end

function svg_header(canvas_size_x, canvas_size_y)
    return """
    <?xml version="1.0" standalone="no"?>
    <!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
    "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
    <svg width="$canvas_size_x" height="$canvas_size_y"
     version="1.1" xmlns="http://www.w3.org/2000/svg">
```

```
    """
end

function pwm_to_logo(source,xpos,ypos,xlen,yscale)
    outstr = ""
    for n in 1:size(source,1)
        outstr *= print_weightvec_at_x_y(source[n,:],xpos+xlen*n,ypos,xlen,yscale)
    end
    outstr *= "\n"
    return outstr
end

function pwmstr_to_io(io,source;log=true)
    log && (source=exp.(source))
    for pos in 1:size(source,1)
        char,color=uni_wvec_params(source[pos,:])
        printstyled(io, char; color=color)
    end
end

function uni_wvec_params(wvec) #assign a single unicode char and color symbol for the most informational position in a position weight vector
    wvec.+=10^-99
    infoscore=(2.0 + sum([x*log(2,x) for x in wvec]))
    infovec = [x*infoscore for x in wvec]
    val,idx=findmax(infovec)
    return char,color=get_char_by_thresh(idx,val)
end
                function get_char_by_thresh(idx,val)
                    char = '?'; seen_thresh=0.
                    for (thresh,threshchar) in thresh_dict
                        val ≥ thresh && thresh ≥ seen_thresh && (char=threshchar; seen_thresh=thresh)
                    end
                    char=='?' && println(val)
                    char=="lwcs" && (char=lwcs_vec[idx])
                    char=="upcs" && (char=upcs_vec[idx])
                    color=cs_vec[idx]

                    return char,color
                end
```

```
function print_weightvec_at_x_y(wvec, xpos, ypos, xlen, yscale)
    # xlen is the length occupied by that column
    # yscale is the total height for 2 information bits i.e. the
    # maximum height available for a "perfect" nucleotide
    outstr = ""
    basestr = "ACGT"
    wvec.+=10^-99  #prevent log(0) = -Inf
    wvec = [x/sum(wvec) for x in wvec] #renorm
    infscore = (2.0 + sum([x*log(2,x) for x in wvec]))
    if infscore==0.0
        return ""
    end
    wvec = [x*infscore*yscale/2.0 for x in wvec]
    # at this point, the sum of all wvec is a maximum of yscale
    wveclist = [(wvec[n],basestr[n]) for n in 1:4]
    wveclist = sort(wveclist)
    curr_ypos = ypos
    for n in 1:4
        curr_ypos -= wveclist[n][1]
        outstr *=
         ↪  print_char_in_rect(wveclist[n][2],xpos,curr_ypos,xlen,wveclist[n][1])
    end
    return outstr
end

blo = 1

function print_char_in_rect(c,x,y,width,height)
    raw_x, raw_y, raw_fontsize, raw_xscale = charvals_dict[c]
    raw_x = (raw_x*raw_xscale-100)/raw_xscale
    raw_y = raw_y-100

    xscale = width/100.0 * raw_xscale
    yscale = height/100.0

    scaled_x = x/xscale + raw_x
    scaled_y = y/yscale + raw_y

```

```
    return "<text x=\"$scaled_x\" y=\"$scaled_y\" font-size=\"$raw_fontsize\" font-weight=\"bold\" font-family=\"Helvetica\" fill=\"rgb"*colour_dict[c]*"\" transform=\"scale($xscale,$yscale)\">$c</text>\n"
end

function svg_footer()
    return "</svg>"
end
```

15.3.16 /src/utilities/ns_progressmeter.jl

```
#UTILITY  REPORTS WORKER NUMBER AND CURRENT ITERATE
mutable struct ProgressNS{T<:Real} <: AbstractProgress
    interval::T
    dt::AbstractFloat
    start_it::Integer
    counter::Integer
    triggered::Bool
    tfirst::AbstractFloat
    tlast::AbstractFloat
    tstp::AbstractFloat
    printed::Bool        # true if we have issued at least one status update
    desc::AbstractString # prefix to the percentage, e.g.  "Computing..."
    color::Symbol        # default to green
    output::IO           # output stream into which the progress is written
    numprintedvalues::Integer   # num values printed below progress in last iteration
    offset::Integer             # position offset of progress bar (default is 0)

    e::IPM_Ensemble
    wm::Worker_Monitor
    tuner::Permute_Tuner
    top_m::ICA_PWM_Model

    mean_stp_time::AbstractFloat

    wk_disp::Bool
    tuning_disp::Bool
```

```
    conv_plot::Bool
    ens_disp::Bool
    lh_disp::Bool
    liwi_disp::Bool
    src_disp::Bool
    no_displayed_srcs::Integer

    disp_rotate_inst::Vector{Any}

    convergence_history::Vector{Float64}

    function ProgressNS{T}(     e::IPM_Ensemble,
                                top_m::ICA_PWM_Model,
                                wm::Worker_Monitor,
                                tuner::Permute_Tuner,
                                interval::T;
                                dt::Real=0.1,
                                desc::AbstractString="Nested Sampling::",
                                color::Symbol=:green,
                                output::IO=stdout,
                                offset::Int=0,
                                start_it::Int=1,
                                wk_disp::Bool=false,
                                tuning_disp::Bool=false,
                                conv_plot::Bool=true,
                                ens_disp::Bool=false,
                                lh_disp::Bool=false,
                                liwi_disp::Bool=false,
                                src_disp::Bool=true,
                                nsrcs::Integer=0,

                                ↪  disp_rotate_inst=[false,0,0,Vector{Vector{Symbol}}
                                ↪  where T
        tfirst = tlast = time()
        printed = false
        new{T}(interval,
         dt,
         start_it,
         start_it,
         false,
         tfirst,
         tlast,
         0.,
```

```
        printed,
        desc,
        color,
        output,
        0,
        offset,
        e,
        wm,
        tuner,
        top_m,
        0.,
        wk_disp,
        tuning_disp,
        conv_plot,
        ens_disp,
        lh_disp,
        liwi_disp,
        src_disp,
        nsrcs,
        disp_rotate_inst,
        zeros(CONVERGENCE_MEMORY))
    end
end

function ProgressNS(e::IPM_Ensemble, wm::Worker_Monitor, tuner::Permute_Tuner, interval::Real; dt::Real=0.1, desc::AbstractString="Nested Sampling::", color::Symbol=:green, output::IO=stderr, offset::Integer=0, start_it::Integer=1, wk_disp::Bool=false, tuning_disp::Bool=false, conv_plot::Bool=false, lh_disp::Bool=false, liwi_disp::Bool=false, ens_disp::Bool=false, src_disp::Bool=false, nsrcs=0, disp_rotate_inst::Vector{Any}=[false,0,0,Vector{Vector{Symbol}}()])
    top_m = deserialize(e.models[findmax([model.log_Li for model in e.models])[2]].path)

    return ProgressNS{typeof(interval)}(e, top_m, wm, tuner, interval, dt=dt, desc=desc, color=color, output=output, offset=offset, start_it=start_it, wk_disp=wk_disp, tuning_disp=tuning_disp, conv_plot=conv_plot, lh_disp=lh_disp, liwi_disp=liwi_disp, ens_disp=ens_disp, src_disp=src_disp, nsrcs=nsrcs, disp_rotate_inst=disp_rotate_inst)
end

```

```julia

function update!(p::ProgressNS, val, thresh; options...)
    p.counter += 1

    p.tstp=time()-p.tlast
    popfirst!(p.tuner.time_history)
    push!(p.tuner.time_history, p.tstp)

    p.interval = val - thresh
    popfirst!(p.convergence_history)
    push!(p.convergence_history, p.interval)

    p.top_m.name ≠ basename(p.e.models[findmax([model.log_Li for model in p.e.models])[2]].path) && (p.top_m=deserialize(p.e.models[findmax([model.log_Li for model in p.e.models])[2]].path))

    updateProgress!(p; options...)
end

function updateProgress!(p::ProgressNS; offset::Integer = p.offset, keep = (offset == 0))
    p.offset = offset
    t = time()
    p.disp_rotate_inst[1] && p.counter%p.disp_rotate_inst[2] == 0 && rotate_displays(p)

    if p.interval ≤ 0 && !p.triggered
        p.triggered = true
        if p.printed
            p.triggered = true
            dur = durationstring(t-p.tfirst)
            msg = @sprintf "%s Converged. Time: %s (%d iterations). logZ: %s\n" p.desc dur p.counter p.e.log_Zi[end]

            print(p.output, "\n" ^ (p.offset + p.numprintedvalues))
            move_cursor_up_while_clearing_lines(p.output, p.numprintedvalues)
            upper_lines=display_upper_dash(p)
            printover(p.output, msg, :magenta)
            lower_lines=display_lower_dash(p)

            p.numprintedvalues=upper_lines + lower_lines + 1
```

```

            if keep
                println(p.output)
            else
                print(p.output, "\r\u1b[A" ^ (p.offset + p.numprintedvalues))
            end
        end
        return
    end

    if t > p.tlast+p.dt && !p.triggered
        p.counter < CONVERGENCE_MEMORY ? mean_step_time=mean(p.tuner.time_history[end-(p.counter-1):end]) :

                                                mean_step_time=mean(p.tuner.time_hist
        msg = @sprintf "%s Iterate: %s Recent step time μ: %s Convergence Interval: %g\n" p.desc p.counter hmss(mean_step_time) p.interval

        print(p.output, "\n" ^ (p.offset + p.numprintedvalues))
        move_cursor_up_while_clearing_lines(p.output, p.numprintedvalues)
        upper_lines=display_upper_dash(p)
        printover(p.output, msg, p.color)
        lower_lines=display_lower_dash(p)

        p.numprintedvalues=upper_lines + lower_lines + 1
        print(p.output, "\r\u1b[A" ^ (p.offset + p.numprintedvalues))

        # Compensate for any overhead of printing. This can be
        # especially important if you're running over a slow network
        # connection.
        p.tlast = t + 2*(time()-t)
        p.printed = true
    end
end
                function rotate_displays(p)
                    curr_inst=p.disp_rotate_inst[3]
                    curr_inst+1>length(p.disp_rotate_inst[4]) ? next_inst=1 : next_inst=curr_inst+1
                    next_disps=p.disp_rotate_inst[4][next_inst]
```

```
169                     for disp in
                         ↪  [:wk_disp,:tuning_disp,:conv_plot,:ens_disp,:lh_disp,:liwi_di
170                         disp in next_disps ?  setproperty!(p, disp, true)
                             ↪  : setproperty!(p, disp, false)
171                     end
172                     p.disp_rotate_inst[3]=next_inst
173                 end
174
175                 function hmss(dt)
176                     dt<0 ? (dt=-dt; prfx="-") : (prfx="")
177                     isnan(dt) && return "NaN"
178                     (h,r) = divrem(dt,60*60)
179                     (m,r) = divrem(r, 60)
180                     (isnan(h)||isnan(m)||isnan(r)) && return "NaN"
181                     string(prfx,Int(h),":",Int(m),":",Int(ceil(r)))
182                 end
183
184                 function display_upper_dash(p::ProgressNS)
185                     wklines = tunelines = cilines = 0
186                     p.wk_disp && (wklines=show(p.output, p.wm,
                         ↪  progress=true);println())
187                     p.tuning_disp && (tunelines=show(p.output, p.tuner,
                         ↪  progress=true);println())
188                     if p.conv_plot
189
                             ↪  ciplot=lineplot([p.counter-(length(p.convergence_history)
                             ↪  p.convergence_history, title="Convergence
                             ↪  Interval Recent History",
                             ↪  xlabel="Iterate",ylabel="CI", color=:yellow)
190                         cilines=nrows(ciplot.graphics)+5
191                         show(p.output, ciplot); println()
192                     end
193                     return wklines + tunelines + cilines
194                 end
195
196                 function display_lower_dash(p::ProgressNS)
197                     lhlines = liwilines = ensemblelines = srclines = 0
198                     if p.lh_disp
199                         lhplot=lineplot(p.e.log_Li[2:end], title="Contour
                             ↪  History", xlabel="Iterate", color=:magenta,
                             ↪  name="Ensemble logLH")
```

```
                        lineplot!(lhplot, [p.e.naive_lh for it in 1:length(p.e.log_Li[2:end])], name="Naive logLH")
                        lhlines=nrows(lhplot.graphics)+5
                        show(p.output, lhplot); println()
                    end
                    if p.liwi_disp && p.counter>2
                        liwiplot=lineplot([max(2,p.counter-(CONVERGENCE_MEMORY-1)), title="Recent iterate evidentiary weight", xlabel="Iterate", name="Ensemble log Liwi", color=:cyan)
                        liwilines=nrows(liwiplot.graphics)+5
                        show(p.output, liwiplot); println()
                    end
                    p.ens_disp && (ensemblelines=show(p.output, p.e, progress=true))
                    p.src_disp && (println("MLE Model Sources:");srclines=show(p.output, p.top_m, nsrc=p.no_displayed_srcs, progress=true))
                    return lhlines + liwilines + ensemblelines + srclines
                end
```

15.3.17 /src/utilities/worker_diagnostics.jl

```
struct Worker_Monitor
    idx::Dict{Integer,Integer}
    persist::BitMatrix
    last_seen::Matrix{Float64}
end

function Worker_Monitor(wk_pool::Vector{<:Integer})
    idx=Dict{Integer,Integer}()
    for (n,wk) in enumerate(wk_pool)
        idx[wk]=n
    end
    persist=trues(1,length(wk_pool))
    last_seen=[time() for x in 1:1, y in 1:length(wk_pool)]
    return Worker_Monitor(idx, persist, last_seen)
end

function update_worker_monitor!(mon,wk,persist)
```

```
    mon.persist[1,mon.idx[wk]]=persist
    mon.last_seen[1,mon.idx[wk]]=time()
end

function Base.show(io::IO, mon::Worker_Monitor; progress=false)
    printstyled("Worker Diagnostics\n", bold=true)
    pers=heatmap(float.(mon.persist), colormap=persistcolor, title="Peristence",labels=false)
    ls=heatmap([time()-ls for ls in mon.last_seen], title="Last Seen",labels=false)
    show(pers)
    println()
    show(ls)
    progress && return nrows(pers.graphics)+nrows(ls.graphics)+7
end

                function persistcolor(z, zmin, zmax)
                    z==1. && return 154
                    z==0. && return 160
                end
```

15.3.18 /test/runtests.jl

```
@info "Loading test packages ..."

using BioMotifInference, BioBackgroundModels, BioSequences, Distributions, Distributed, Random, Serialization, Test
import StatsFuns: logsumexp, logaddexp
import BioMotifInference:estimate_dirichlet_prior_on_wm, assemble_source_priors, init_logPWM_sources, wm_shift, permute_source_weights, get_length_params, permute_source_length, get_pwm_info, get_erosion_idxs, erode_source, init_mix_matrix, mixvec_decorrelate, mix_matrix_decorrelate, most_dissimilar, most_similar, revcomp_pwm, score_sources_ds!, score_sources_ss!, weave_scores_ss!, weave_scores_ds!, IPM_likelihood, consolidate_check, consolidate_srcs, pwm_distance, permute_source, permute_mix, perm_src_fit_mix, fit_mix, random_decorrelate, reinit_src, erode_model, reinit_src, shuffle_sources, accumulate_mix, distance_merge, similarity_merge, info_fill, converge_ensemble!, reset_ensemble!, Permute_Tuner, PRIOR_WT, TUNING_MEMORY, CONVERGENCE_MEMORY, tune_weights!, update_weights!, clamp_pvec!
import Distances: euclidean
```

```

@info "Beginning tests ... "
using Random
Random.seed!(786)
O=1000;S=50

include("pwm_unit_tests.jl")
include("mix_matrix_unit_tests.jl")
include("likelihood_unit_tests.jl")
include("consolidate_unit_tests.jl")
include("permute_func_tests.jl")
include("permute_tuner_tests.jl")
include("ensemble_tests.jl")
@info "Tests complete!"
```

15.3.19 /test/spike_recovery.jl

```
#Testbed for synthetic spike recovery from example background

using nnlearn, BGHMM, HMMBase, Distributions, Random, Serialization,
↪ Distributions

Random.seed!(786)

#CONSTANTS
no_obs=1000
obsl=140

```

```
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])])

struc_sig=[.1 .7 .1 .1
           .1 .1 .1 .7
           .1 .7 .1 .1]
periodicity=8
struc_frac_obs=.35

tata_box=[.05 .05 .05 .85
          .85 .05 .05 .05
          .05 .05 .05 .85
          .85 .05 .05 .05
          .425 .075 .075 .425
          .85 .05 .05 .05
          .425 .075 .075 .425]
tata_frac_obs=.7
tata_recur_range=1:4

combined_ensemble = "/bench/PhD/NGS_binaries/nnlearn/combined_ensemble"

#JOB CONSTANTS
const position_size = 141
const ensemble_size = 100
const no_sources = 3
const source_min_bases = 3
const source_max_bases = 12
@assert source_min_bases < source_max_bases
const source_length_range= source_min_bases:source_max_bases
const mixing_prior = .3
```

```
@assert mixing_prior ≥ 0 && mixing_prior ≤ 1
const models_to_permute = ensemble_size * 3

job_sets=[
([
    ("PS", (no_sources)),
    ("PM", (no_sources)),
    ("PSFM", (no_sources)),
    ("PSFM", (no_sources*10, .8, 1.)),
    ("FM", ()),
    ("DM", (no_sources)),
    ("SM", (no_sources)),
    ("RD", (no_sources)),
    ("RI", (no_sources)),
    ("EM", (no_sources))
],[.025, .025, .025, .025, .775, .025, .025, .025, .025, .025]),
([
    ("PS", (no_sources)),
    ("PM", (no_sources)),
    ("PSFM", (no_sources)),
    ("PSFM", (no_sources, 0., 1.)),
    ("PSFM", (no_sources, 0.8, .0)),
    ("FM", ()),
    ("DM", (no_sources)),
    ("SM", (no_sources)),
    ("RD", (no_sources)),
    ("RI", (no_sources*10)),
    ("EM", (no_sources))
],[.05, .15, .15, .10, .05, .20, 0.0, 0.05, 0.15, 0.10, .0]),
]
job_limit=4

const prior_wt=1.1

#FUNCTIONS
function setup_obs(hmm, no_obs, obsl)
    obs=vec(Int64.(rand(hmm,obsl)[2]))
    for o in 2:no_obs
        obs=hcat(obs, vec(Int64.(rand(hmm,obsl)[2])))
    end
    obs=vcat(obs,zeros(Int64,1,no_obs))
    return obs
end
```

```

function spike_irreg!(obs, source, frac_obs, recur)
    truth=Vector{Int64}()
    for o in 1:size(obs,2)
        if rand()<frac_obs
            push!(truth, o)
            for r in rand(recur)
                rand()<.5 && (source=nnlearn.revcomp_pwm(source))
                pos=rand(1:size(obs,1)-1)
                pwm_ctr=1
                while pos⩽size(obs,1)-1&& pwm_ctr⩽size(source,1)
                    obs[pos,o]=rand(Categorical(source[pwm_ctr,:]))
                    pos+=1
                    pwm_ctr+=1
                end
            end
        end
    end
    return truth
end

function spike_struc!(obs, source, frac_obs, periodicity)
    truth=Vector{Int64}()
    truthpos=Vector{Vector{Int64}}()
    for o in 1:size(obs,2)
        if rand()<frac_obs
            push!(truth, o)
            rand()<.5 && (source=nnlearn.revcomp_pwm(source))
            pos=rand(1:periodicity)
            posvec=Vector{Int64}()
            while pos⩽size(obs,1)
                pos_ctr=pos
                pwm_ctr=1
                while pos_ctr⩽size(obs,1)-1&&pwm_ctr⩽size(source,1)
                    push!(posvec,pos_ctr)
                    base=rand(Categorical(source[pwm_ctr,:]))
                    obs[pos_ctr,o]=base
                    pos_ctr+=1
                    pwm_ctr+=1
                end
                pos+=periodicity+pwm_ctr
            end
            push!(truthpos,posvec)
```

```
        end
    end
    return truth,truthpos
end

function get_BGHMM_lhs(obs,hmm)
    lh_mat=zeros(size(obs,1)-1,size(obs,2))
    for o in 1:size(obs,2)
        obso=zeros(Int64,size(obs,1),1)
        obso[1:end,1] = obs[:,o]
        lh_mat[:,o]=BGHMM.get_BGHMM_symbol_lh(Matrix(transpose(obso)),hmm)
    end
    return lh_mat
end

@info "Setting up synthetic observation set..."
obs=setup_obs(hmm, no_obs, obsl)
struc_truth,struc_postruth=spike_struc!(obs, struc_sig, struc_frac_obs, periodicity)
irreg_truth=spike_irreg!(obs, tata_box, tata_frac_obs, tata_recur_range)

@info "Calculating background likelihood matrix..."
bg_lhs=get_BGHMM_lhs(obs,hmm)

@info "Assembling source priors..."
#prior_array= [struc_sig, tata_box]
prior_array= Vector{Matrix{Float64}}()
source_priors = nnlearn.assemble_source_priors(no_sources, prior_array, prior_wt, source_length_range)

@info "Assembling ensemble..."
path=randstring()
isfile(string(path,'/',"ens")) ? (ens = deserialize(string(path,'/',"ens"))) :
    (ens = nnlearn.IPM_Ensemble(path, ensemble_size, source_priors, (falses(0,0), mixing_prior), bg_lhs, obs, source_length_range))


job_set_thresh=[-Inf,ens.naive_lh]
param_set=(job_sets,job_set_thresh,job_limit)

@info "Converging ensemble..."
```

```
nnlearn.ns_converge!(ens, param_set, models_to_permute, .0001,
↪  model_display=no_sources, backup=(true,5))

rm(path,recursive=true)

#811973
```