GPCell: A Performant Framework for Gaussian Processes in Bioinformatics

Tristan Sones-Dykes

MMath Mathematics

April 2025

School of Mathematics and Statistics

The University of St Andrews

Submitted in total fulfilment of the requirements of the degree of MMath Mathematics

Abstract

Gene expression regulation is pivotal in cellular function, with significant advancements since the 1960s. Notably, Jacob and Monod's work elucidated gene activation mechanisms in response to external stimuli via mRNA transcription modulation (Jacob & Monod, 1961). Subsequent research, such as Hardin *et al.* (1990)'s study on circadian rhythms in *Drosophila melanogaster*, highlighted oscillatory gene expression through protein-mediated RNA inhibition. Building upon these foundations, Phillips *et al.* (2017) investigated gene expression patterns in neural progenitor cells, identifying correlations between oscillatory behavior and differentiation. Their methodology employed Gaussian processes to classify gene expression time series using MATLAB.

This dissertation extends their approach by developing a Python library that facilitates Gaussian process fitting and oscillation detection across diverse datasets. Enhancements include an extensible modelling framework, allowing for the easy addition of fitting techniques like MCMC; being based coherently on top of Tensorflow Probability, taking advantage of computational advancements and giving access to a suite of priors, model types, and optimisers; and an automated Continuous Integration/Continuous Deployment (CI/CD) pipeline, with accuracy tests for models and automatically generated docs (Sones-dykes, 2025).

Future works can now prioritise scientific discovery and model choice, using a suite of utilities that simplify the model-fitting process.

Table of contents

1 Introduction

1.1 Biological Background

Oscillatory gene expression is a widespread phenomenon across diverse biological systems, serving critical roles in timing and information encoding. Examples span a vast range of timescales: from ultradian calcium oscillations on the order of seconds, to cell-cycle and developmental rhythms of a few hours, up to 24-hour circadian clocks; Phillips *et al.* (2017). For instance, circadian gene expression in individual fibroblast cells is self-sustained – each cell functions as an autonomous oscillator passing its phase to offspring cells; Nagoshi *et al.* (2004). Similarly, oscillatory dynamics have been observed in the NF- B signaling pathway (pulsatile nuclear localization driving gene bursts; Nelson *et al.* (2004)) and the p53 tumor suppressor system (repeated p53 protein pulses after DNA damage; Geva-Zatorsky et al., 2006 journals.plos.org). During vertebrate embryonic development, the segmentation clock exemplifies how oscillations pattern multicellular systems: waves of gene expression (Hairy/Hes genes) sweep across the presomitic mesoderm, coordinating somite formation in space and time; *Patterning embryos with oscillations:* Oates *et al.* (2012). These and other cases (e.g. cyclic expression of cell-fate regulators like HES1/Ascl1 in neural progenitors) highlight that oscillatory gene expression is a fundamental, conserved mechanism in biology.

Biologically, oscillations confer several functional advantages. Intuitively, oscillatory circuits can act as temporal regulators or "clocks" like those mentioned above. Beyond time-keeping, oscillatory dynamics allow information encoding in ways that static levels cannot. Because an oscillation is characterized by parameters like frequency and amplitude, cells can modulate these features to encode signals journals.plos.org . A well-known example is the ERK/Ras pathway, where the frequency vs. amplitude of ERK pulses can differentially activate downstream genes; Sonnen & Aulehla (2014). In stem cells, the pattern of gene expression (oscillatory versus sustained) can determine cell fate decisions. For instance, neural stem cells exhibit 2–3 hour oscillations in HES1, which in turn cause out-of-phase oscillations in the proneural factor Ascl1. Oscillatory Ascl1 expression keeps these cells in a proliferative, undifferentiated state, whereas switching to sustained Ascl1 expression triggers differentiation into neurons; Imayoshi *et al.* (2013). Thus, beyond gene expression level alone, the *dynamics* of expression carry biologically relevant information that can dictate outcomes; Marinopoulou *et al.* (2021).

1.1.1 Mechanistic basis

At the molecular level, oscillatory gene expression typically arises from negative feedback loops with delays. A canonical motif is a transcriptional repressor that inhibits its own expression after a time lag, producing rhythmic ups and downs. The HES1 oscillator is a classic example: HES1 protein represses the Hes1 gene, but protein turnover creates a delay that allows transcript levels to rise and fall periodically; Marinopoulou *et al.* (2021). This delayed negative feedback mechanism was predicted by theoretical models – Goodwin (1965) – and later observed experimentally; Hirata *et al.* (2002). Synthetic biology has also demonstrated that simple gene circuits can oscillate; *the Repressilator* Elowitz & Leibler (2000) engineered in E. coli was a landmark showing that a three-gene feedback loop yields oscillatory protein expression, validating design principles of biological oscillators. Mathematically, these systems are often described by limit cycle oscillators or coupled differential equations with delays, and analysis tools from nonlinear dynamics (e.g. Hopf bifurcation analysis) have been applied to understand their stability and periodicity – Novák & Tyson (2008) – which states all biochemical oscillators are characterised by negative feedback with time delay. Stochastic effects, however, play a major role at the single-cell level – gene expression involves small numbers of mRNAs/proteins reacting stochastically, leading to intrinsic noise in dynamics; Phillips *et al.* (2017). In particular, transcriptional bursting (episodic production of mRNA) can produce fluctuations that mimic or obscure oscillatory patterns, or cell might exhibit irregular, quasi-periodic bursts rather than a perfect periodic sinusoid; this blurring of "signal" (true oscillation) and "noise" (aperiodic fluctuation) makes it challenging to decide if a given single-cell time series is genuinely oscillatory; Phillips *et al.* (2017).

1.1.2 Single-cell perspective

Until recently, gene expression oscillations were primarily characterized in cell populations or tissue averages, which can obscure cell-to-cell differences. Advances in single-cell genomics and imaging have revolutionized this area by enabling time-resolved measurements in individual cells. Live-cell reporters (e.g. luciferase or fluorescent proteins under control of oscillatory promoters) allow continuous recording of gene expression in single cells over hours or days Phillips *et al.* (2017). This has revealed profound cell-to-cell heterogeneity: even in genetically identical cells, some may oscillate strongly while others do not, or oscillations may vary in period and amplitude from cell to cell journals.plos.org . For example, in a population of fibroblasts, each cell's circadian phase can drift, leading to desynchronized averages despite robust single-cell cycles; Welsh *et al.* (2004), Nagoshi *et al.* (2004). In the case of HES1 dynamics, live single-cell imaging showed only a subset of cells oscillate measurably, and oscillation coherence can change with developmental context Phillips *et al.* (2017). Single-cell RNA sequencing (scRNA-seq) and single-molecule FISH (Kwon (2013)) have provided complementary "snapshots" of gene expression across many individual cells. The surge of such data in the last decade has created a need for quantitative methods to analyze noisy time series from individual cells.

1.2 Current Work

Traditional signal-processing approaches for periodicity detection (Fourier transforms, autocorrelation, Lomb–Scargle periodograms, etc.) often fail on short, noisy cellular time series with irregular oscillation profiles. These classical methods assume long, stationary signals or low noise, conditions rarely met in single-cell experiments (which may only track a few oscillation cycles before photobleaching or cell division, and where noise is significant). To address this, researchers have turned to statistical modeling approaches that can explicitly account for noise and uncertainty. In particular, Gaussian processes (GPs) have emerged as a powerful framework for analyzing oscillatory time series in single cells. GPs are flexible non-parametric models that can capture arbitrarily complex temporal patterns with well-characterized uncertainty, making them attractive for classifying oscillations in noisy data. Phillips *et al.* (2017) introduced a pioneering GP-based method to decide if a given single-cell trajectory is oscillatory or not. Their approach combined a mechanistic stochastic model of a gene regulatory oscillator with GP regression, enabling an objective classification that outperformed the Lomb–Scargle periodogram. In tests on simulated data and on live-cell imaging of a luminescent Hes1 reporter, the GP method reliably distinguished truly oscillatory cells from those with mere noise-driven fluctuations. Their method is, however, not fully Bayesian as it uses parametric bootstrapping to obtain a better estimate of the classification boundary.

Choice of programming environment is a key consideration when developing a Gaussian Process (GP) modeling framework for bioinformatics. Historically, MATLAB was widely used for GP research for example the influential GPML toolbox accompanying Rasmussen & Williams (2005), and the later GPstuff package; Vanhatalo *et al.* (n.d.), provided a rich set of GP algorithms in MATLAB. However, in recent years the balance has shifted strongly toward Python for both research and practical applications. Below, we justify the decision to use Python (and specifically a Python-based GP library) over MATLAB for our GPCell framework, considering ecosystem maturity, performance, extensibility, and integration needs:

1. Open-Source Ecosystem and Reproducibility: Python is free and open-source, which fosters broad usage and community-driven development. Anyone can run and inspect Python code without restrictive licenses, an important factor for reproducible science. By contrast, MATLAB is proprietary software – requiring a license – which can hamper reproducibility and accessibility (Ince *et al.*, 2012).

2. Specialized GP Libraries and Performance: The Python ecosystem for GPs is more mature and performant than MATLAB's current offerings. Notably, **GPflow** – Matthews *et al.* (n.d.) – and **GPyTorch** – Gardner *et al.* (n.d.) – are two leading libraries that leverage modern machine learning frameworks for speed and scalability. GPflow builds on TensorFlow, enabling automatic differentiation and accelerated linear algebra on GPUs.

3. Ecosystem and Integration with Bioinformatics Tools: Bioinformatics workflows often involve diverse data types (genomic sequences, expression matrices, network data) and multiple analysis steps. Python has become a lingua franca in data science, enabling seamless integration of GP modeling with upstream and downstream analyses. For instance, one can use pandas or NumPy to manipulate genomic data, feed it into a GP model from GPflow, and then visualize results with Matplotlib or Seaborn, all within one environment. There are also domain-specific Python libraries (Scanpy for single-cell RNA-seq, Biopython, scikit-learn, etc.) that interoperate well. In contrast, MATLAB, while strong in matrix computations, is less commonly used in genomics and lacks the breadth of specialized bioinformatics libraries. Many cutting-edge bioinformatics methods (e.g. for single-cell data or deep learning-based analyses) are released in Python or R, not MATLAB, making Python a more natural choice for compatibility. Additionally, Python's ability to wrap C/C++ and interface with R (via rpy2) means it can serve as a hub, combining methods across ecosystems – something more cumbersome with MATLAB. You could also use reticulate and connect Python to an R script, using it as a computational backend.

4. Continuous Integration and Deployment (CI/CD) Friendliness: Developing a robust software package like GPCell benefits from modern DevOps practices. Python's packaging system (pip/conda) and testing frameworks (unittest, pytest) allow easy distribution and validation of the code on multiple platforms. Free CI services (GitHub Actions, Travis CI, etc.) can automatically run test suites on each commit, which is feasible since Python is open source.
Additionally, high performance MATLAB code, including GPML that Phillips *et al.* (2017) based their library on, requires a separate compile and build process before anything is ran. This added complexity, unlike with exclusively Python dependencies, reduces the number of researchers it is able to target, though it may not seem so to the developers.

1.3 Problem Statement

This presents a need for an extensible, generalisable library to easily handle Gaussian Processes and classify gene expressions into oscillatory and non-oscillatory using modern development techniques such as unit tests, consistent typing, and CI/CD. As well as a gap in the literature for a fully Bayesian approach that is at least as good as the parametric bootstrap approach; or can offer some additional benefits of Bayesian inference the current one is unable to.

2 Methods

2.1 Gaussian Processes

2.1.1 Introduction

A Gaussian Process (GP) is a powerful, non-parametric Bayesian approach to modeling distributions over functions. In regression tasks, GPs provide a flexible framework that not only predicts mean function values but also quantifies uncertainty, making them particularly suitable for modeling noisy and complex biological time-series data, such as gene expression profiles.

Formally, a GP is defined as a collection of random variables, any finite number of which have a joint Gaussian distribution. A GP is fully specified by its mean function $m(\mathbf{x})$ and covariance function (kernel) $k(\mathbf{x}, \mathbf{x}')$:

$$f(\mathbf{x}) \sim GP(m(\mathbf{x}), k(\mathbf{x}, \mathbf{x}')) \tag{2.1}$$

For practical applications, and in our case, the mean function is often assumed to be zero ($m(\mathbf{x}) = 0$) as we can remove trends from our data and then focus on the kernel component which differentiates our models.

2.1.2 Regression

Given a set of training (for us, time) inputs $\mathbf{X} = \mathbf{x}_1, ..., \mathbf{x}_n$ and observations $\mathbf{Y} = \mathbf{y}_1, ..., \mathbf{y}_n$; each observation is modelled as $\mathbf{y}_i = f(\mathbf{x}_i) + \epsilon_i$ with $\epsilon_i \sim N(0, \sigma_n^2)$ Gaussian noise. The objective is to predict the value $f(\mathbf{x}_*)$ at new input $\mathbf{x}_*$.

The joint distribution of the observed values and function at $\mathbf{x}_*$ is given by:

$$\begin{bmatrix} \mathbf{y} \\ f_* \end{bmatrix} \sim \mathcal{N}(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} \mathbf{K}(\mathbf{X}, \mathbf{X}) + \sigma_n^2 I & \mathbf{K}(\mathbf{X}, \mathbf{x}_*) \\ \mathbf{K}(\mathbf{x}_*, \mathbf{X}) & \mathbf{K}(\mathbf{x}_*, \mathbf{x}_*) \end{bmatrix}) \tag{2.2}$$

Where $\mathbf{K}(\mathbf{X}, \mathbf{X})$ is the covariance matrix computed over the training inputs and $\mathbf{K}(\mathbf{X}, \mathbf{x}_*)$ is covariance vector between the training inputs and the new input.

The predictive distribution for f_* is then Gaussian with known mean and variance.

2.1.3 Kernels

The choice of kernel $k(\mathbf{x}, \mathbf{x}')$ is crucial as it determines the behaviour of the model. In our case, we will be training models with periodic and aperiodic kernels, then assessing their quality of fits to determine whether or not the underlying gene expression is oscillating or not.

By modelling gene expression time-series generally using the Chemical Master Equation, deriving the Linear Noise Approximation as in Elf & Ehrenberg (2003), and assuming the deterministic steady-state has been reached, Phillips *et al.* (2017) shows that the underlying biological system can be modelled using an Ornstein Uhlenbeck (OU) process.

Thus, to create a pair of periodic and aperiodic kernels, they can take the OU kernel as it is already aperiodic and augment the OU kernel with a cosine kernel to create a quasi-periodic oscillatory process.

$$\mathbf{K}_{OU}(\tau) = \sigma_{OU} \exp(-\alpha\tau) \tag{2.3}$$

$$\mathbf{K}_{OUosc}(\tau) = \sigma_{OU} \exp(-\alpha\tau) \cos(\beta\tau) \tag{2.4}$$

Such that, if the OUosc model has a significantly better fit on a trace despite added model complexity, then it can be reasonably concluded that the trace is oscillatory.

2.1.4 Inference

Closed-Form Posterior and Marginal Likelihood

Gaussian Process regression with a Gaussian likelihood admits an exact posterior, which is itself a GP characterized by an analytically tractable mean and covariance function; Rasmussen & Williams (2005).

The marginal likelihood (evidence) of the observations under the GP prior can be computed in closed form and differentiated with respect to hyperparameters, enabling Type II Maximum Likelihood (ML) estimation – empirical Bayes – where kernel parameters are set by maximizing the log marginal likelihood.
Gradient-based optimization (e.g., using L-BFGS or conjugate gradients) is standard practice, as the derivatives of the log marginal likelihood with respect to hyperparameters can be expressed in terms of the inverse and determinant of the kernel matrix.

Approximate Inference

When the likelihood is non-Gaussian, usually in classification or count data, exact GP inference is intractable, necessitating approximate Bayesian inference.
These also come into play when we need to numerically integrate the posterior distribution, in our case for calculating the Bayes Factor.

There are many methods here, but we will go over two major ones in GP inference. Variational Inference and Markov Chain Monte Carlo.

1. Markov Chain Monte Carlo (MCMC)

MCMC methods, such as Hamiltonian Monte Carlo (HMC) or No-U-Turn Sampler (NUTS), draw samples from the joint posterior over latent functions and hyperparameters, yielding a full Bayesian treatment with asymptotic exactness; Titsias *et al.* (2011).
Although more computationally intensive, MCMC does not rely on Gaussian approximations and is widely used when full uncertainty quantification is critical; Titsias *et al.* (2008).

2. Variational Inference

Variational approaches posit a tractable family and optimize a variational lower bound (the evidence lower bound, ELBO) on the log marginal likelihood. Sparse variational methods introduce inducing variables to reduce computational cost, leading to scalable GP models such as the Sparse Variational Gaussian Process; Titsias (2009).

To summarise, while Type II ML provides efficient point estimates of hyperparameters, fully Bayesian approaches use MCMC to marginalize over hyperparameters, capturing posterior uncertainty and avoiding overfitting Titsias *et al.* (2008). The choice between these reflects a trade-off between computational tractability (favoring optimization) and comprehensive uncertainty quantification (favoring sampling).

2.2 Bayesian Classification Method

We investigate Bayesian classification methods to classify single-cell gene expression time series as oscillatory or non-oscillatory. This approach builds upon the methodology of Phillips *et al.* (2017), but we extend it by using Bayesian inference (via MCMC) for hypothesis testing and uncertainty estimation. The goal is to determine, for each single-cell time course, whether there is statistically significant evidence of an oscillatory pattern as opposed to aperiodic noise. Below, we describe the modeling setup, prior choices, inference procedure, and model comparison techniques we used.

2.2.1 Overall pipeline

Our method follows mostly Phillips *et al.* (2017) until the bootstrapping section. We have calculated the background noise and detrended the input cell traces, then created multiple replicate models for each cell that have been initialised to their maximum likelihood solution (recommended in GPflow docs; "MCMC (markov chain monte carlo) — GPflow 2.9.1 documentation" (2025)). However, we have only fit the OUosc kernel models (2.4), as they are all that is needed for the Bayes factor calculation.

Following this, we sample their chains with burn-in, creating posterior samples. Then we pool the traces and assess their convergence using the Gelman-Rubin $\hat{R}$ and Effective Sample Size (ESS); Gelman & Rubin (1992), Kong (1992).
Finally, we calculate the Bayes factor using the MCMC samples and classify into oscillatory and non-oscillatory with a cutoff.

2.2.2 Model Selection and Classification

The next step is to identify the better model and thus, whether or not each trace is oscillatory. In order to do this we calculate the Bayes factor.

The problem setup is as follows. We consider $n \in \mathbb{N}$ competing models and are calculating (relative) plausibility of model $\mathcal{M}_i$ $(i = 1, ..., n)$, given the prior probability model and observations $\mathbf{Y}$.
For this, the posterior model probability $p(\mathcal{M}_i|\mathbf{Y})$, can be expanded using Bayes formula and the law of total conditional probability.

$$p(\mathcal{M}_i|\mathbf{Y}) = \frac{p(\mathbf{Y}|\mathcal{M}_i)p(\mathcal{M}_i)}{p(\mathbf{Y})} = \frac{p(\mathbf{Y}|\mathcal{M}_i)p(\mathcal{M}_i)}{\sum_{j=1}^{m} p(\mathbf{Y}|\mathcal{M}_1)p(\mathcal{M}_i)} \tag{2.5}$$

Where the denominator is the sum of the marginal likelihoods times the prior of all n models.
Taking only two models, as in our scenario, we start to uncover the Bayes factor equation. We can calculate the relative posterior model plausibility of $\mathcal{M}_1$ to $\mathcal{M}_2$ by taking the ratio of the posterior probabilities of both models, also known as the posterior odds; details are in Berger & Molina (2005).

$$\frac{p(\mathcal{M}_1|\mathbf{Y})}{p(\mathcal{M}_2|\mathbf{Y})} = \frac{p(\mathbf{Y}|\mathcal{M}_1)p(\mathcal{M}_1)}{\sum_{j=1}^{2} p(\mathbf{Y}|\mathcal{M}_i)p(\mathcal{M}_i)} \times \frac{\sum_{j=1}^{2} p(\mathbf{Y}|\mathcal{M}_i)p(\mathcal{M}_i)}{p(\mathbf{Y}|\mathcal{M}_2)p(\mathcal{M}_2)} \tag{2.6}$$

$$= \underbrace{\frac{p(\mathcal{M}_1)}{p(\mathcal{M}_2)}}_{\text{prior odds}} \times \underbrace{\frac{p(\mathbf{Y}|\mathcal{M}_1)}{p(\mathbf{Y}|\mathcal{M}_2)}}_{\text{Bayes factor}} \tag{2.7}$$

This shows that our posterior odds are a product of two factors, the prior model odds and the Bayes factor. The Bayes factor is the standard method for hypothesis testing in the Bayesian framework and, as we are using Bayesian inference methods that enable drawing samples from the joint posterior, we can take advantage of this after the fact.

Savage-Dickey Ratio

Originally described in Dickey (1971) and explained in-depth in Wagenmakers *et al.* (2010); the Savage-Dickey ratio is a method for efficiently calculating the Bayes factor of nested models, requiring only that the support of the prior distribution of the model-difference parameter – δ – in the larger model includes the null value of that parameter δ_0.
These requirements allow it to sidestep the requirement for full marginal integration, requiring only posterior samples of the target parameter; it does this by taking the ratio of the posterior to prior density at the null value of the model-difference parameter:

$$BF_{12} = \frac{p(\mathbf{Y}|\mathcal{M}_1)}{p(\mathbf{Y}|\mathcal{M}_2)} = \frac{p(\delta = \delta_0|\mathbf{Y}, \mathcal{M}_1)}{p(\delta = \delta_0|\mathcal{M}_1)} \tag{2.8}$$

This is applicable to our problem, as the non-oscillatory kernel equals the oscillatory kernel at $\beta = 0 \implies cos(\beta\tau) = 1$, and because the prior for that parameter includes $\beta = 0$. So, the Bayes factor is comparing the posterior density of the lengthscale parameter at 0 to the prior density at 0.

We implement this by sampling β as part of the MCMC under the oscillatory model In practice, the prior is continuous so the density at exactly 0 is theoretical; we approximate this either by a standard average on the posterior samples, or by calculating a posterior kernel density estimate and evaluating that on our interval.

Bridge Sampling

Bridge sampling is a flexible Monte Carlo method to approximate marginal likelihoods. Originally introduced in Bennett (1976) and formalized by Meng & Wong (1996), bridge sampling constructs an optimal “bridge” function between two densities – typically the posterior and a convenient proposal – to minimize estimator variance; Gronau *et al.* (2017).

In its basic form, given the parameter of interest θ's samples of from two corresponding normalised densities $p_i(\theta) = q_i(\theta)/c_i$ $(i = 1, 2)$, the following identity holds:

$$r \equiv \frac{c_1}{c_2} = \frac{E_2\left[q_1(\theta)\alpha(\theta)\right]}{E_1\left[q_2(\theta)\alpha(\theta)\right]} \tag{2.9}$$

Where $\alpha(\theta)$ a suitably chosen bridge function. Bennett's original work framed this in terms of free-energy differences in physics, then Meng & Wong (1996) extended it to general normalizing-constant estimation in statistics.
To then compute the Bayes factor, there are two options:

1. Two-step Approach

A straightforward strategy is to apply bridge sampling separately to each model to obtain their posterior marginal densities, then form their ratio. This is what is currently implemented in GPCell.

In this, p_1 is the fit GP posterior $p(\theta|\mathbf{Y})$ and p_2 is a normalised multivariate normal approximation to the posterior (so $c_2 = 1$). The bridge identity (2.9) then specialises to:

$$\hat{p}(\mathbf{Y}) = c_1 = \frac{\frac{1}{n_2}\sum_{i=1}^{n_2} q_1(\omega_i)\alpha(\omega_i)}{\frac{1}{n_1}\sum_{i=1}^{n_1} q_2(\phi_i)\alpha(\phi_i)}$$

Where $\{\omega_i\}$ are draws from p_2 and $\{\phi_i\}$ are draws from the posterior GP. In practice, the original GP posterior samples are split into two groups; one of which is used to fit the multivariate normal approximation, which then generates
A detailed explanation and derivation is available in Meng & Schilling (2002).

2. Direct Ratio Estimation

Alternatively, bridge sampling can directly estimate the ratio r, corresponding to the marginal likelihoods of $\mathcal{M}_1$ and $\mathcal{M}_2$ without separate evaluations; Meng & Schilling (2002).
By treating the posterior distributions under each model as un-normalized densities, one constructs a single bridge estimator for r, which is exactly the Bayes factor B_{12}.

Computational Considerations

While the runtimes of these algorithms are quite fast, there is very little in terms of public implementation for bridge sampling in Python. Gronau *et al.* (2020) provides an R library for bridge sampling, and could be integrated into GPCell using `reticulate` if needed; this is exactly what Elsemüller *et al.* (2023) did to use bridge sampling as one of their metrics in a similar setup: *A Deep Learning Method for Comparing Bayesian Hierarchical Models.*

Lao (2017) is a single short Jupyter Notebook containing a `pymc` implementation, using functions and methods long since deprecated. As this looks like one of the last general, public implementations, I translated it to modern `pymc`; this can now be used to validate Python methods using other inference backends (like GPflow or GPyTorch).

2.2.3 Prior choices

Our models are formulated in the same way as in Phillips *et al.* (2017); however, due to needing being used for the Savage-Dickey ratio and MCMC sampling, we cannot use the same priors.
This is because, in the maximum likelihood estimation method, they are just used to generate starting points for the gradient descent. Using the bayes factor has many advantages, like quantifying uncertainty and allowing for positive and negative hypothesis testing; however, the added complexity in the result makes it far more dependent on the prior distribution, which is shown in Sinharay & Stern (2002) and many other studies.

The β prior needs to have a support of $[0, +\infty)$, to make the models nested, and they need to be positive as bridge sampling relies on unconstrained chains. Phillips *et al.* (2017) used tight, uniform priors for evaluation on the simulated dataset; in order to match these for fair evaluation against their method, we quantile-matched smooth priors onto the intervals of the original uniform ones.
For σ and α, that had uniform priors on $[0.1, 2.0]$, they have no requirement of support at 0 (as they aren't our parameter of interest) therefore a log-normal distribution was chosen with the specification of 95% falling within the interval.

If $X \sim \text{LogNormal}(\mu, \sigma^2)$, then $\ln X \sim \mathcal{N}(\mu, \sigma^2)$. The 2.5% and 97.5% quantiles of $\mathcal{N}(\mu, \sigma^2)$ are $\mu \pm 1.96\sigma$, hence set:

$$\ln(a) = \mu - 1.96\sigma, \ \ln(b) = \mu + 1.96\sigma$$

Which we solve to obtain:

$$\mu = \frac{\ln(a) + \ln(b)}{2} \text{ and } \sigma = \frac{\ln(b) - \ln(a)}{2 \times 1.96}$$

The oscillatory parameter β, originally uniform on $[0.1, 4.0]$, must include 0 in its support, so a half-normal distribution was chosen. $X \sim \text{HalfNormal}(\sigma^2)$.
It only has one parameter to tune, so with the condition that 97.5% of its mass lies above the lower bound of its original underlying prior:

$$P(X \leq b) = \text{erf}(\frac{b}{\sigma\sqrt{2}}) = 0.95 \implies \sigma = \frac{b}{\sqrt{2}\text{erf}^{-1}(0.95)}$$

This may seem like quite an extreme condition; however, once at very small values, the sampler struggles to escape the local maximum of fitting to the null model. This was the reasoning behind Phillips *et al.* (2017) choosing those lower bounds in their original method.

But, a low prior density near $\beta = 0$ means the Savage-Dickey ratio, which relies on the kernel density estimate at the null value, is expected to struggle without a large number of samples.
This presents a trade-off between trace convergence and accuracy that may mean bridge sampling is the better choice, especially the direct ratio estimation.
Wang *et al.* (2019) puts forward Warp Bridge sampling, specifically Warp-U transformations. These allow the target and proposal distributions p_1 and p_2 to be much more opposed, with Warp-U transformations able to map a multi-modal density to a uni-modal one.

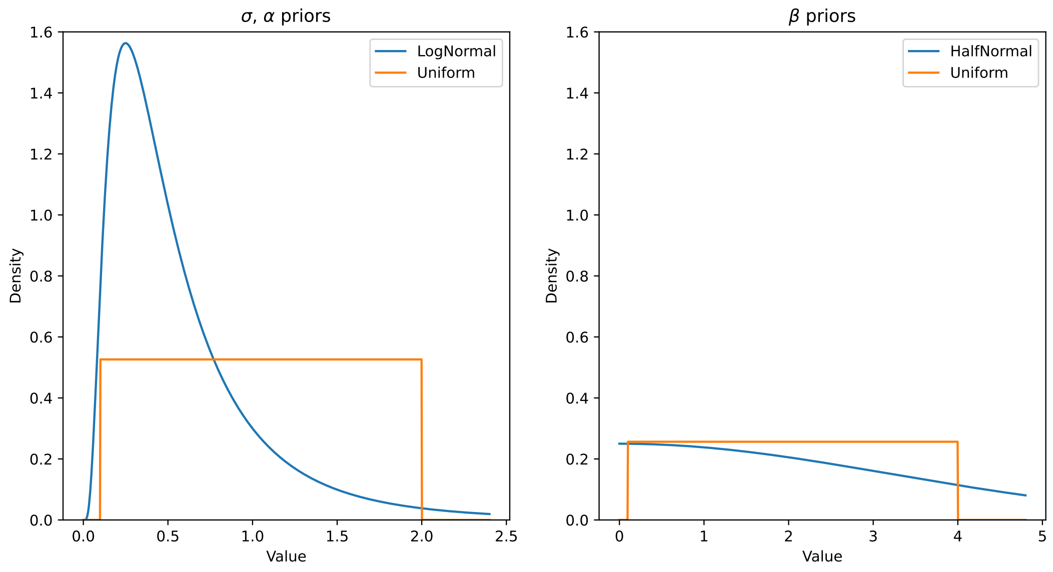

Figure 2.1: Plots of the MCMC prior distributions used instead of the original ML priors.

These were designed to slightly favour weight towards small values, as the sampler needs to be able to reach those values to provide accurate marginal calculations, especially for the Savage-Dickey ratio.

2.3 GPCell

Utilizing the GPflow library, which is based on TensorFlow, GPCell provides a user-friendly interface for researchers, with an `OscillatorDetector` class specifically designed for Phillips *et al.* (2017)'s, and similar, use-cases. It also has unit tests to verify correctness, a strong type system to aid development and upkeep, and a suite of general utility functions that automate the model fitting process, using a multiprocessing pipeline to increase fitting speed.

2.3.1 Software Architecture

GPCell is structured into clearly defined, modular components. A schematic class diagram of the main components – `OscillatorDetector`, `GaussianProcess`, `GPRConstructor`, and supporting utility modules – is below:

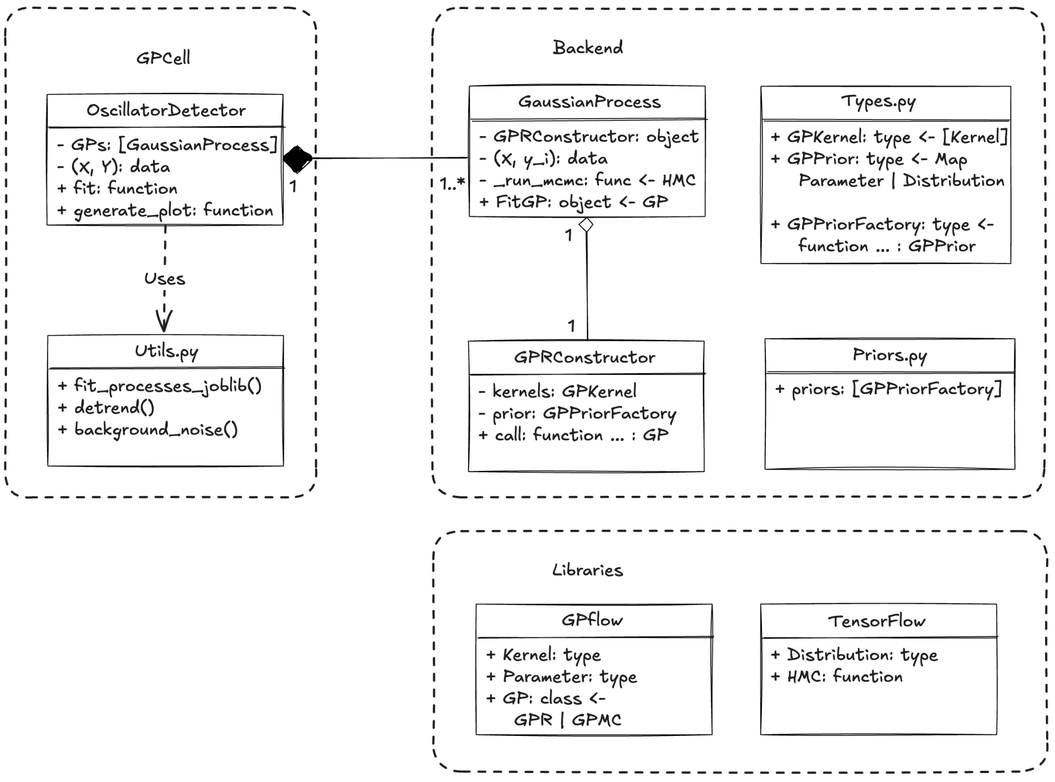

Figure 2.2: Class diagram of GPCell

`GaussianProcess`: An interface for GPflow's Gaussian Process models. It encapsulates the logic for optimising the given model type, predicting on new data, and extracting hyperparameters and values from the kernel. It is instantiated with a `GPRConstructor`, which defines the inference type (i.e: MLE vs MCMC), kernel, priors, and trainable hyperparameters.

This provides a simple interface for users, allows for memory optimisation by only using the posterior data of the fit model after training, and contains individual-process utilities like plotting the fit model and common fit model administration tasks.

`GPRConstructor`**:** This class dynamically instantiates GPflow GP models, which require data, kernels, priors, and trainable hyperparameter flags.

It is separated from `GaussianProcess`, as creating a model and attaching the above is simpler in one step than to construct a `GaussianProcess` with a model type and priors, and then feed in the data dynamically to fit. This is largely due to the structure of GPflow and having to interact with TensorFlow's compute graph.
These can then be given to multiple GPs being fit in parallel on different CPU processes, making parallelism simpler and allowing `GaussianProcess` objects to be reused on new data with dynamically generated priors.

`OscillatorDetector`**:** Serving as the main user-facing class for oscillation classification, it orchestrates the GP modelling workflow. It takes raw input data, performs preprocessing steps (background noise fitting and detrending), fits oscillatory and non-oscillatory models, and classifies gene expression traces based on statistical criteria such as Bayesian Information Criterion (BIC) or Log-Likelihood Ratios (LLRs).

As well as being a self-contained usable product of the library, it can also be seen as an example of how one can use the library to do analysis.
A similarly performant, equivalent analysis could be done in a notebook using just the functions in `utils.py` and the users choice of priors and kernels from GPflow. Alternatively, a variety of analyses and extensions could be made to fit and model Gaussian Processes for different use-cases.

`Utils.py`**:** Containing optimised functions such as `fit_processes_joblib` and `get_time_series` (runs Gillespie simulations of the modelled reactions in parallel), the utilities module supports parallel processing via Joblib, facilitating the rapid fitting and generation of large datasets.

2.3.2 Optimisations and Computational Strategies

The computational complexity associated with GP model fitting – typically $O(n^3)$ – necessitates optimised implementations to handle large bioinformatics datasets efficiently. GPCell employs several optimisation strategies:

Parallel Processing: GPCell exploits the parallel processing capabilities provided by `Joblib`. Functions like `fit_processes_joblib` provide general and simple to use utilities that parallelise core functionality. This function significantly reduces compute time by parallelising the fitting of multiple Gaussian Process models across the available CPU cores, yielding substantial performance improvements.

This feature shows some of the big advantages of using Python over MATLAB for this use-case. `Joblib` only has Python dependencies, making it easier to use than the MATLAB library which requires external compilation for the parallelism to work.

Additionally, Joblib's backend loky can pickle or serialise more complex objects, as well as those defined in notebooks, allowing for a more user-friendly interface and efficient encapsulation.
It also means that, when expanding objects like GaussianProcess, users don't have to also add serialisation logic which would be required to store the additional variables required for MCMC inference.

As for performance advantages, Joblib supports cached functions across multiple processes, so those used to run Gillespie simulations can be compiled once and then distributed.
It also provides simple access to Numpy's optimised memmaps, meaning the input time series matrix can be temporarily written to the disk with multiple processes accessing it in parallel, instead of having to give each process its own copy of the experimental/simulated data. As fitting Gaussian Processes requires multiple processes accessing small subsets of the same large file, it is a good fit for memmaps.
Using memmaps also improves the stability of the parallel processes which is vital for large statistical experiments.

Just-In-Time Compilation: For the computationally intensive Gillespie simulations, GPCell utilises Numba, a Just-In-Time compiler for Python. This enables the conversion of Python code (exclusively written in base Python and Numpy) into optimised machine code at runtime.

This is a perfect use-case for JIT compilation, as the simulation functions themselves are not very complicated computationally, they just need to be ran sequentially for a lot of iterations.

2.3.3 Extensibility and Modularity

GPCell's design strongly emphasises extensibility, allowing researchers to customise various components to their specific requirements.
This is enabled by a strong type system integrated throughout the library and the large TensorFlow (Probability) backend.

Custom Kernels: GPCell supports the creation and combination of user-defined kernels, facilitating a variety of modelling goals in biology [find other GPs used in bio/bioinf]. Users can readily implement new kernels or combine existing ones using arithmetic operations, enhancing model flexibility.

```python
# Kernel types
GPKernel = Union[Type[Kernel], Sequence[Type[Kernel]]]
"""
Type for Gaussian Process kernels.
Can be a single kernel or a list for a composite kernel.
If a list is provided, an operator can be provided or `*` is used.
"""
GPOperator = Callable[[Kernel, Kernel], Kernel]
"""
Type for operators that combine multiple kernels.
```

```
Examples for a product `*` operator:
    - `operator.mul` (what is used as default)
    - `gpflow.kernels.Product`
    - `lambda a, b: a * b`
"""
```

Here, `Kernel` is the base class for GPflow's kernels, and is just an extension of TensorFlow's `Module` class which is used to build models in TensorFlow generally.

The code shows that kernels (for use in `GPRConstructor`) can either be a single kernel or a list of kernels, and that the operator that combines them – if multiple are given – is just a callable that takes two kernels and outputs one regardless of operation.
In this format, K_{OUosc} is defined as:

```
# Kernels and operator for a GP model
ouosc_kernel = [Matern12, Cosine]
ouosc_op = operator.mul  # which is the default and typically omitted
```

To create a new kernel, one inherits from `Kernel` – or one of its subclasses like `IsotropicStationary`; alternatively, take any class/TensorFlow `Module` and make it compatible with the `Kernel` interface.
This is the definition of `Matern12` in GPflow as an example.

```
# Kernel example
class Matern12(IsotropicStationary):
    """
    The kernel equation is
    k(r) = sigma² * exp{-r}
    """

    @check_shapes(
        "r: [batch..., N]",
        "return: [batch..., N]",
    )
    def K_r(self, r: TensorType) -> tf.Tensor:
        return self.variance * tf.exp(-r)
```

It uses GPflow's `IsotropicStationary` kernel base class, which is for stationary kernels that only depend on the Euclidean distance $r = ||\mathbf{x}' - \mathbf{x}||_2$, requiring only the implementation of $k(r)$ or $k(r^2)$, $k(r) = k(\mathbf{x}', \mathbf{x})$.

Prior Distributions: Through `GPRConstructor`, researchers can easily define and adjust priors for GP hyperparameters, allowing more precise control over model behaviour and incorporating domain-specific knowledge effectively. This is shown in the typing of `GPPrior` and `GPPriorFactory` in `types.py`:

```python
# Prior types
GPPrior = Mapping[str, Union[Parameter, Numeric]]
"""
Type for Gaussian Process priors.

The keys are the paths to the priors, e.g `.kernel.lengthscales`
or `.kernel.kernels[1].variance`.
Values:
    - `Parameter` used for transformed parameters e.g: Softplus.
    - `Numeric` used for numeric initial values.
"""
GPPriorFactory = Callable[..., GPPrior]
"""
Type for Gaussian Process prior factories.

Used for defining a pattern that will dynamically generate priors
for each GPR model.
"""
```

This gives a clear, type-checked definition of a parameter of `GPRConstructor` (`GPPriorFactory`). That being, a mapping (dictionary) from a string describing its position in the kernel, to either a GPflow `Parameter` – which can be any transformed number for MLE, or a TensorFlow Probability prior distribution for MCMC – or `Numeric`, an unconstrained Python or `Numpy` number.
This information is all available in the docstring, so is easy to find whilst coding and also gets exported to the auto-generated documentation; there are also examples of each in `backend/priors.py`.

Inference Techniques: The modular nature of GPCell allows straightforward integration of different inference techniques, ranging from classical maximum likelihood estimation (MLE) to more complex Bayesian approaches such as variational inference or MCMC.

MCMC via TensorFlow Probability: As an example custom/additional inference technique, MCMC functionality has been added to the library just by adding code in two objects – `GPRConstructor` then `GaussianProcess`:

```python
# GPRConstructor
match self.inf:
    case "MCMC":
        likelihood = Gaussian()
        model = GPMC((X, y), kernel, likelihood)
```

This is a simple addition to the `self.inf` check that instantiates a GPflow `GPMC` (MCMC) model instead of a `GPR` (MLE) model, with the rest of the object being the same.

```
# GaussianProcess
match gp_reg:
    case GPR():
        # Keep the posterior for predictions
        self.fit_gp = gp_reg.posterior()
    case GPMC():
        (
            self.fit_gp,
            self.samples,
            self.parameter_samples,
            self.param_to_name,
            self.name_to_index,
        ) = self._run_mcmc(gp_reg, sampler=self.mcmc_sampler)
```

This directly matches against the model type, improving safety and reducing the number of extra parameters needed.
Earlier in the `GaussianProcess` fit method, both `GPR` and `GPMC` models are optimised to their MLE positions. This means that `GPR` models are already fit and only the posterior is taken, whereas the `GPMC` is then ran through either a Hamiltonian Monte Carlo (HMC) or No-U-Turn Sampler (NUTS) implemented in TensorFlow Probability, according to `sampler`.

3 Results

3.1 Reproduces Previous Results

A key plot in Phillips *et al.* (2017) is the ROC curve plot comparing the GP method to the Lomb-Scargle method. It is reproduced below, with added AUC values for both methods.

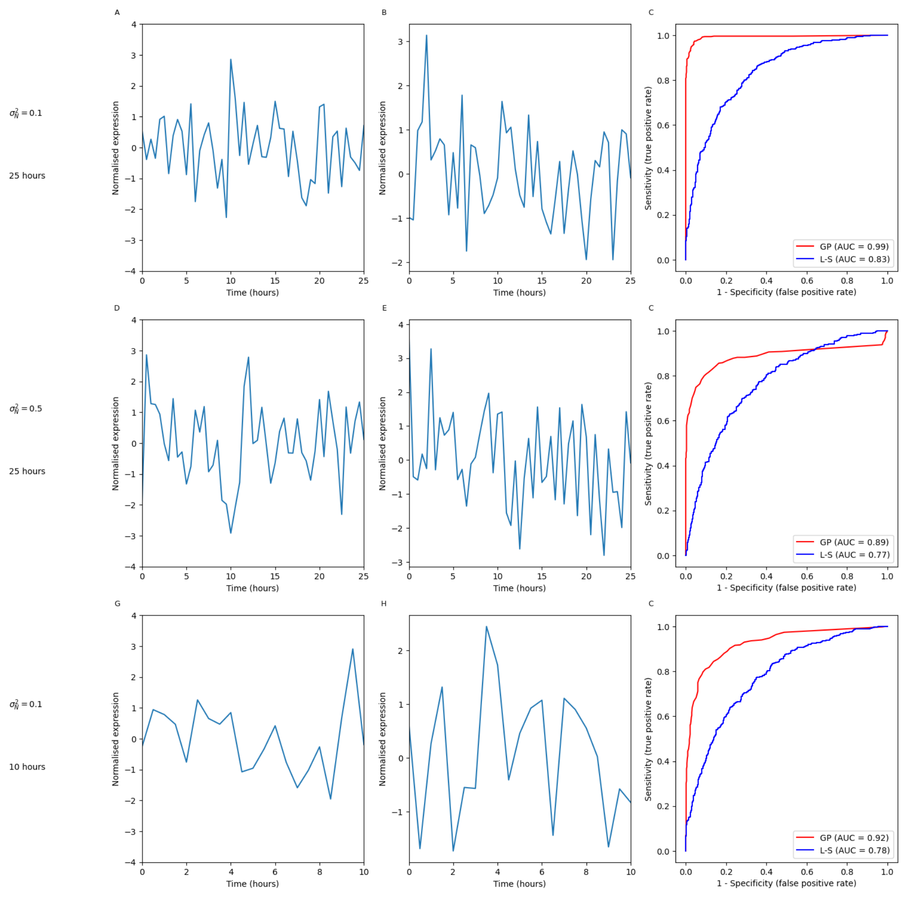

Figure 3.1: Example simulated traces and model ROC curves, factored by simulated noise level σ^2 and trace length (time)

This shows that GPCell fits the same quality models as the MATLAB implementation, on MATLAB generated data. The ROC curves don't have margins on the original plot, adding them makes spotting discrepancies between the models easier.

The added AUC values give a lot of information; despite the $\sigma^2 = 0.5$ ROC for the GP method looking far worse than the others, the AUC shows us it is, quantitatively, not as dissimilar from the others as it looks.

3.2 Performance Improvements

There were two key areas of performance improvement, in fitting models and in simulating data. Some of the improvement was just from switching to MATLAB, but the majority, especially on large datasets, comes from deliberate computational improvements.

3.2.1 Model Fitting

Using `Joblib`, GPCell is able to distribute the job of fitting a set of replicate models to a trace, as processes ran on separate cores.
Shown below are the results of fitting the core BIC pipeline (OU and OUosc models) to homogeneous (square/matrix shaped), and non-homogeneous (jagged) datasets. This makes a difference as, through `Joblib`, GPCell is programmed to take advantage of NumPy's `memmap` when possible. It writes the square dataset to a temporary file and provides each process with a read-only reference, allowing concurrent access and reducing memory overhead.

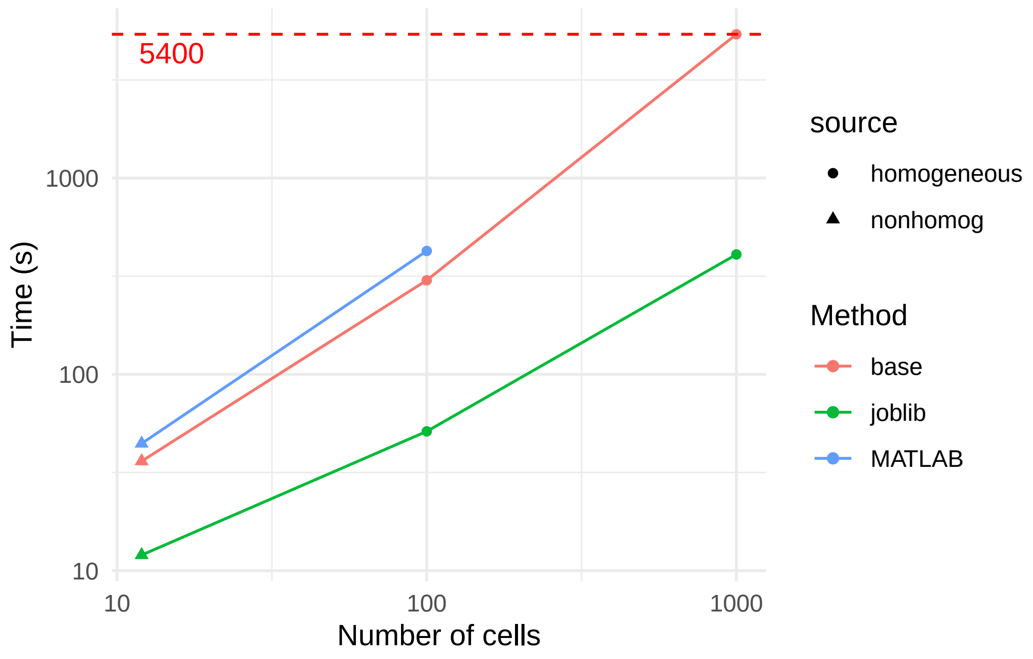

Figure 3.2: Time taken to fit processes, on a log scale, split by data source and backend. Base Python at 1000 cells reached 90 minutes before being stopped.

This shows the stark difference between fitting models in parallel and in sequence as the number of cells increases.

The datasets for the tests on the larger numbers of cells are all homogeneous. This is because the parallel fitting is more stable when the dataset is able to be made into an optimised `memmap`.
Note that although it seems like a difficult limit to require the input traces to be homogeneous for the best performance, the performance and stability of the algorithm without it is still a large improvement.

Additionally, during Phillips *et al.* (2017)'s bootstrap step, the synthetic cell generations are homogeneous by definition; this reduces the time to complete such a compute-intensive task massively. If you are only classifying 12 single-cell traces like in their Hes example dataset, that turns into fitting 12 $\times$ 10 (simulations) $\times$ 10 (replicates) = 1200 models, this is easily programmed and processed in parallel with GPCell.

3.2.2 Data Generation

In the original MATLAB implementation, parallel processing is only available with external tooling, and does not take full advantage of vector processing.
Below is a table comparing the time taken to simulate cells (needed for model evaluation and tuning) at the different levels of optimisation.

Table 3.1: Average data simulation times

Implementation	Time..s.
Python, parallel and njit	58.64
Python, parallel	315.23
Python	3722.16
MATLAB	Approx 1.5 hours

Each optimisation decreases the time taken by an order of magnitude, the final optimisation only being possible to due compiling the simulation function (which requires it only be written in NumPy with limited base Python).

These optimisations help in two ways; first by increasing the number of experiments that can be ran and reducing the impact on servers, but also by lowering the computational barrier to use and contribute to the library effectively.

3.3 MCMC Support

Show MCMC models being fit through GPCell and other metrics.

Show performance gains also make it to MCMC.

3.4 Bayesian Classifier

Show pymc model fits and plots

Discuss the results of Savage-Dickey vs bridge sampling

4 Discussion

Concise overview of the library's features and benefits (performance, reproducibility, modularity)

4.1 Comparison with Existing Methods

Strengths and improvements over previous MATLAB-based methods (performance, extensibility, user accessibility)

Benchmarking results explicitly compared

4.2 Challenges and Limitations

Issues encountered (e.g., parameter tuning, hyperparameter priors, scalability to large datasets)

Addressing previous inconsistent choices (priors selection and impact)

4.3 Future Directions

Potential enhancements (Variational inference methods, GPU support, web-based interactive version)

Broader implications for the bioinformatics community and reproducibility

Final thoughts summarising key findings, relevance, and impact of your work

References

Bennett, C.H. (1976) Efficient estimation of free energy differences from monte carlo data. *Journal of Computational Physics*, **22**, 245–268.

Berger, J.O. & Molina, G. (2005) Posterior model probabilities via path-based pairwise priors. *Statistica Neerlandica*, **59**, 3–15.

Dickey, J.M. (1971) The weighted likelihood ratio, linear hypotheses on normal location parameters. *The Annals of Mathematical Statistics*, **42**, 204–223.

Elf, J. & Ehrenberg, M. (2003) Fast Evaluation of Fluctuations in Biochemical Networks With the Linear Noise Approximation. *Genome Research*, **13**, 2475–2484.

Elowitz, M.B. & Leibler, S. (2000) A synthetic oscillatory network of transcriptional regulators. *Nature*, **403**, 335–338.

Elsemüller, L., Schnuerch, M., Bürkner, P.-C., & Radev, S.T. (2023) A deep learning method for comparing bayesian hierarchical models.

Gardner, J.R., Pleiss, G., Bindel, D., Weinberger, K.Q., & Wilson, A.G. (n.d.) GPyTorch: Blackbox matrix-matrix gaussian process inference with GPU acceleration, DOI: 10.48550/arXiv.1809.11165.

Gelman, A. & Rubin, D.B. (1992) Inference from iterative simulation using multiple sequences. *Statistical Science*, **7**, 457–472.

Goodwin, B.C. (1965) Oscillatory behavior in enzymatic control processes. *Advances in Enzyme Regulation*, **3**, 425–437.

Gronau, Q.F., Sarafoglou, A., Matzke, D., Ly, A., Boehm, U., Marsman, M., Leslie, D.S., Forster, J.J., Wagenmakers, E.-J., & Steingroever, H. (2017) A tutorial on bridge sampling. *Journal of Mathematical Psychology*, **81**, 80–97.

Gronau, Q.F., Singmann, H., & Wagenmakers, E.-J. (2020) bridgesampling: An R Package for Estimating Normalizing Constants. *Journal of Statistical Software*, **92**, 1–29.

Hardin, P.E., Hall, J.C., & Rosbash, M. (1990) Feedback of the Drosophila period gene product on circadian cycling of its messenger RNA levels. *Nature*, **343**, 536–540.

Hirata, H., Yoshiura, S., Ohtsuka, T., Bessho, Y., Harada, T., Yoshikawa, K., & Kageyama, R. (2002) Oscillatory expression of the bHLH factor Hes1 regulated by a negative feedback loop. *Science*, **298**, 840–843.

Imayoshi, I., Isomura, A., Harima, Y., Kawaguchi, K., Kori, H., Miyachi, H., Fujiwara, T., Ishidate, F., & Kageyama, R. (2013) Oscillatory control of factors determining multipotency and fate in mouse neural progenitors. *Science*, **342**, 1203–1208.

Jacob, F. & Monod, J. (1961) Genetic regulatory mechanisms in the synthesis of proteins. *Journal of Molecular Biology*, **3**, 318–356.

Kong, A. (1992) A note on importance sampling using standardized weights.

Kwon, S. (2013) Single-molecule fluorescence in situ hybridization: Quantitative imaging of single RNA molecules. *BMB Reports*, **46**, 65–72.

Lao, J. (2017) Marginal Likelihood in Python and PyMC3.

Marinopoulou, E., Biga, V., Sabherwal, N., Miller, A., Desai, J., Adamson, A.D., & Papalopulu, N. (2021) HES1 protein oscillations are necessary for neural stem cells to exit from quiescence. *iScience*, **24**, 103198.

Matthews, A.G. de G., Wilk, M. van der, Nickson, T., Fujii, K., Boukouvalas, A., León-Villagrá, P., Ghahramani, Z., & Hensman, J. (n.d.) GPflow: A gaussian process library using TensorFlow, DOI: 10.48550/arXiv.1610.08733.

MCMC (markov chain monte carlo) — GPflow 2.9.1 documentation (2025).

Meng, X.-L. & Schilling, S. (2002) Warp Bridge Sampling. *Journal of Computational and Graphical Statistics*, DOI: 10.1198/106186002457.

Meng, X.-L. & Wong, W.H. (1996) SIMULATING RATIOS OF NORMALIZING CONSTANTS VIA a SIMPLE IDENTITY: A THEORETICAL EXPLORATION. *Statistica Sinica*, **6**, 831–860.

Nagoshi, E., Saini, C., Bauer, C., Laroche, T., Naef, F., & Schibler, U. (2004) Circadian gene expression in individual fibroblasts: Cell-autonomous and self-sustained oscillators pass time to daughter cells. *Cell*, **119**, 693–705.

Nelson, D.E., Ihekwaba, A.E.C., Elliott, M., Johnson, J.R., Gibney, C.A., Foreman, B.E., Nelson, G., See, V., Horton, C.A., Spiller, D.G., Edwards, S.W., McDowell, H.P., Unitt, J.F., Sullivan, E., Grimley, R., Benson, N., Broomhead, D., Kell, D.B., & White, M.R.H. (2004) Oscillations in NF- B signaling control the dynamics of gene expression. *Science*, **306**, 704–708.

Novák, B. & Tyson, J.J. (2008) Design principles of biochemical oscillators. *Nature Reviews Molecular Cell Biology*, **9**, 981–991.

Oates, A.C., Morelli, L.G., & Ares, S. (2012) Patterning embryos with oscillations: Structure, function and dynamics of the vertebrate segmentation clock. *Development*, **139**, 625–639.

Phillips, N.E., Manning, C., Papalopulu, N., & Rattray, M. (2017) Identifying stochastic oscillations in single-cell live imaging time series using Gaussian processes. *PLOS Computational Biology*, **13**, e1005479.

Rasmussen, C.E. & Williams, C.K.I. (2005) *Gaussian Processes for Machine Learning*. The MIT Press.

Sinharay, S. & Stern, H.S. (2002) On the sensitivity of bayes factors to the prior distributions. *The American Statistician*, **56**, 196–201.

Sones-dykes, T. (2025) GPCell documentation.

Sonnen, K.F. & Aulehla, A. (2014) Dynamic signal encoding—from cells to organisms. *Seminars in Cell & Developmental Biology*, mRNA stability and degradation & timing of mammalian embryogenesis & positioning of the mitotic spindle, **34**, 91–98.

Titsias, M. (2009) Variational learning of inducing variables in sparse gaussian processes. *Journal of Machine Learning Research - Proceedings Track*, **5**, 567–574.

Titsias, M.K., Rattray, M., & Lawrence, N.D. (2011) Markov chain monte carlo algorithms for gaussian processes. (Cemgil, A.T., Barber, D., & Chiappa, S. eds). Cambridge: Cambridge University Press, pp. 295–316.

Titsias, M., Lawrence, N., & Rattray, M. (2008) Markov chain monte carlo algorithms for gaussian processes. *Inference and Estimation in Probabilistic Time-Series Models*, 9.

Vanhatalo, J., Riihimäki, J., Hartikainen, J., Jylänki, P., Tolvanen, V., & Vehtari, A. (n.d.) Bayesian modeling with gaussian processes using the GPstuff toolbox, DOI: 10.48550/arXiv.1206.5754.

Wagenmakers, E.-J., Lodewyckx, T., Kuriyal, H., & Grasman, R. (2010) Bayesian hypothesis testing for psychologists: A tutorial on the savage–dickey method. *Cognitive Psychology*, **60**, 158–189.

Wang, L., Jones, D.E., & Meng, X.-L. (2019) Warp bridge sampling: The next generation, DOI: 10.48550/arXiv.1609.07690.

Welsh, D.K., Yoo, S.-H., Liu, A.C., Takahashi, J.S., & Kay, S.A. (2004) Bioluminescence imaging of individual fibroblasts reveals persistent, independently phased circadian rhythms of clock gene expression. *Current biology: CB*, **14**, 2289–2295.

List of figures

List of tables