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 (**gpcell_docs?**).

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

The regulation of gene expression is fundamental to cellular processes, dictating how genes are activated or repressed in response to various stimuli. A landmark discovery by Jacob and Monod in the 1960s (Jacob & Monod, 1961) demonstrated that genes could be switched on or off through the modulation of mRNA transcription, a mechanism that garnered them the Nobel Prize. While their research provided insights into gene regulation mechanisms, it did not delve into the temporal dynamics of gene expression.

In the 1980s, research into the temporal aspects of gene expression gained momentum. Hardin *et al.* (1990) proposed the Transcription Translation Negative Feedback Loop to explain circadian rhythms in *Drosophila melanogaster.* They discovered that certain genes exhibited oscillatory behavior, regulated by proteins that inhibited their own mRNA production, thereby establishing a feedback loop.

1.2 Problem Statement

Furthering this line of inquiry, Phillips *et al.* (2017) explored gene expression patterns in neural progenitor cells, revealing that such expression could be either oscillatory or aperiodic. Notably, oscillatory expression was associated with cellular differentiation. Their methodology involved fitting both oscillating and non-oscillating Gaussian processes to gene expression time series data, enabling the classification of genes based on their oscillatory behavior.

However, their approach was primarily developed for a specific biological context and model setup, limiting its extensibility. It was also developed using a library no longer being developed in MATLAB, narrowing its possible scope, as well as requiring users to obtain a license and follow an unfamiliar build pipeline (compared to that of R or Python). Additionally, their technique is not fully bayesian, as it uses a parametric bootstrap to fine-tune the classification boundary.

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 as good or better than the parametric bootstrap approach.

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) $k(\mathbf{x}, \mathbf{x}')$:

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

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:

$$\begin{bmatrix} \mathbf{y} \\ f_* \end{bmatrix} \sim \mathcal{N}(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} \mathbf{K}(\mathbf{X}, \mathbf{X}) + \sigma_n^2 I & \mathbf{K}(\mathbf{X}, \mathbf{x}_*) \\ \mathbf{K}(\mathbf{x}_*, \mathbf{X}) & \mathbf{K}(\mathbf{x}_*, \mathbf{x}_*) \end{bmatrix}) \tag{2.1}$$

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.

$$\mathbf{K}_{OU}(\tau) = \sigma_{OU} \exp(-\alpha\tau) \tag{2.2}$$

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

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.

2.2 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.

2.2.1 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:

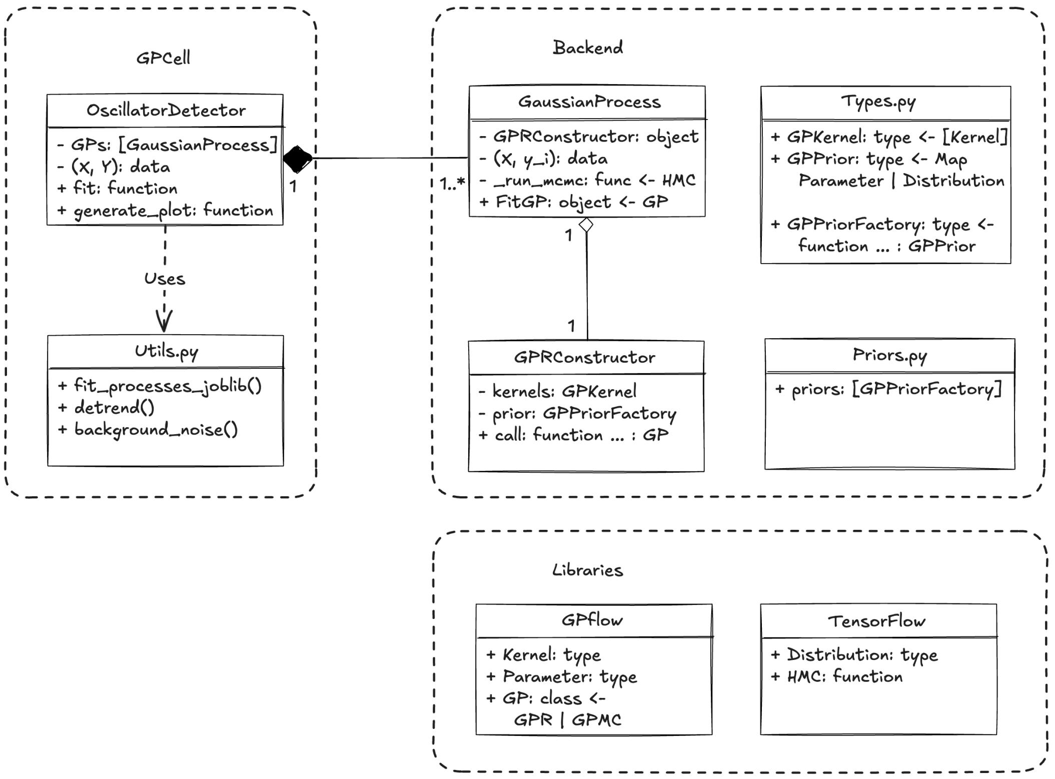

Figure 2.1: Class diagram of GPCell

`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.

2.2.2 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.

2.2.3 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.

```
# 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)
```

This 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`:

```
# 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.

```python
# 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 Reproducibility of Prior Results

ROC figures comparing your library against previous MATLAB implementation (both GP and Lomb-Scargle methods)

Explicit statistical comparison (AUC values)

3.2 Performance Analysis

Figures and tables showing speed improvements (e.g., parallelisation with Joblib, simulation with numba)

Discussion on computational resources needed (RAM, CPU/GPU)

3.3 MCMC Implementation

Example MCMC fits (trace plots, parameter distributions)

Discuss implications and accuracy improvements from MCMC inference

3.4 Practical Demonstration and Extensibility

Showcase practical use-cases (example time-series, how results are interpreted)

Short code snippets demonstrating extensibility (showing actual examples using your library)

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

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

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.

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

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.

List of figures

List of tables