# Pints: A Python Package for Picking Probable Parameters

**Michael Clerx**
David J. Gavaghan
Sanmitra Ghosh
Ben C. Lambert
Chon Lok Lei
Gary R. Mirams
Martin Robinson

UNIVERSITY OF OXFORD

# Background

- In many domains, we encounter the problem of **parametrising** mechanistic models (e.g. of biological systems) using **noisy time-series data**

# Background

- In many domains, we encounter the problem of **parametrising** mechanistic models (e.g. of biological systems) using **noisy time-series data**
- **PINTS** stands for Probabilistic Inference on Noisy Time-Series

# Background

- In many domains, we encounter the problem of **parametrising** mechanistic models (e.g. of biological systems) using **noisy time-series data**
- **PINTS** stands for Probabilistic Inference on Noisy Time-Series
- It is a Python-based tool to tackle this problem within a **probabilistic framework**

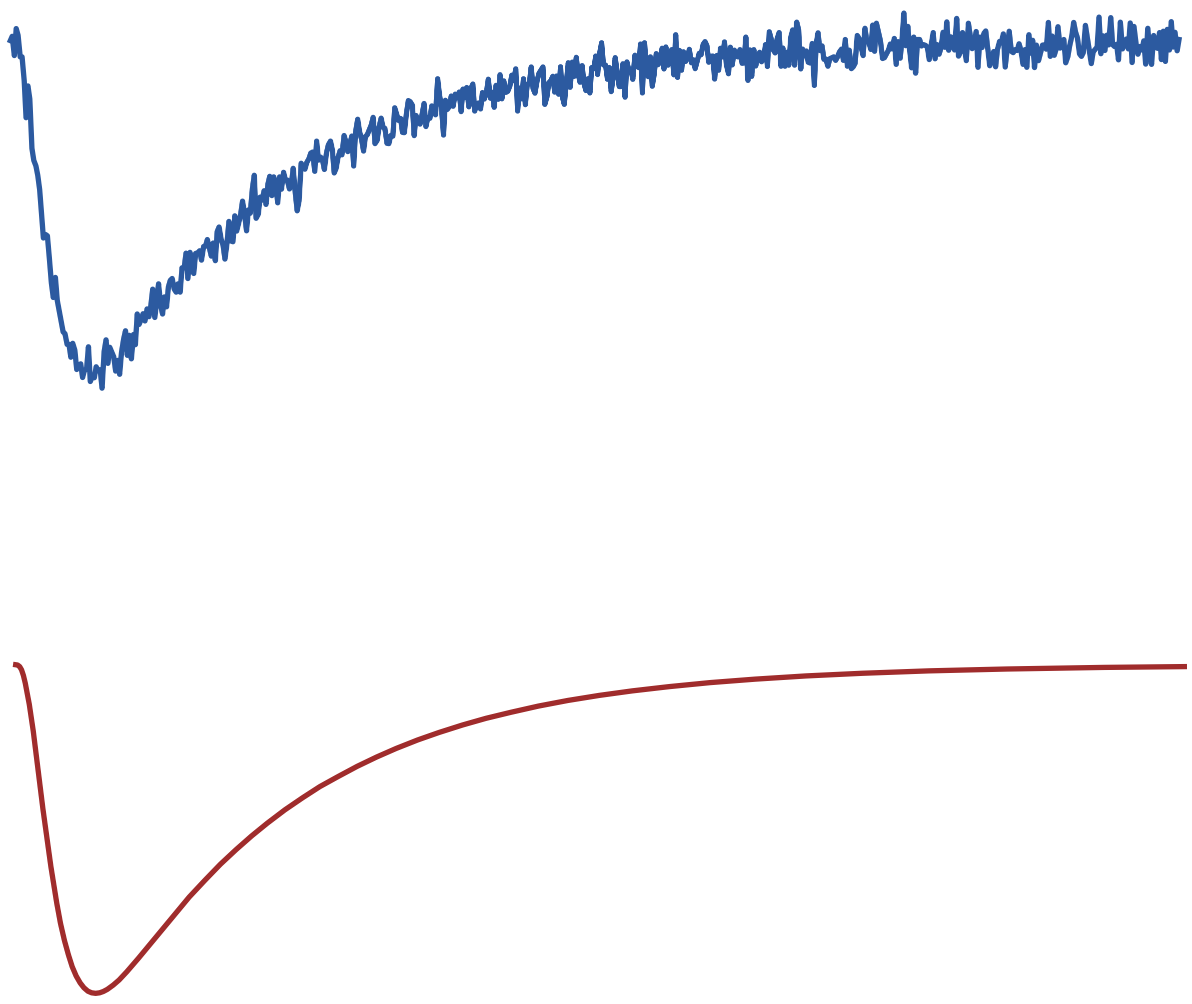
# Background

- Free Pints!

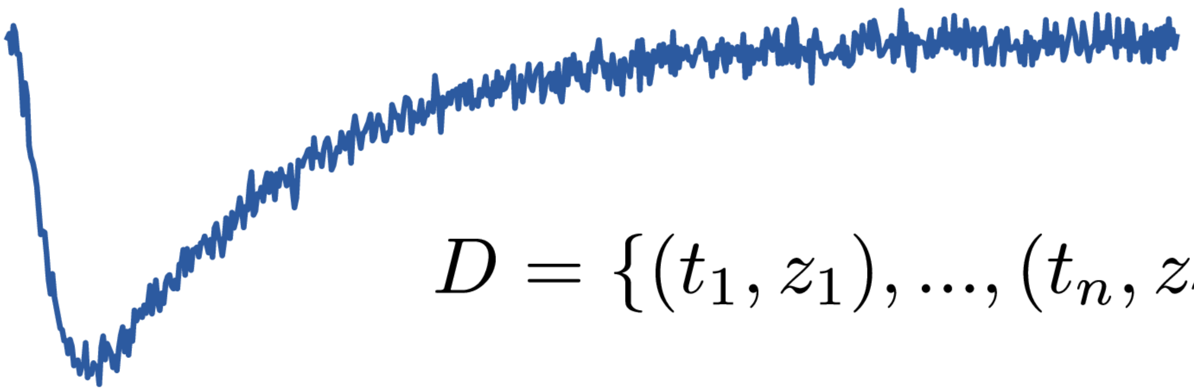
https://github.com/pints-team/pints

- **Pints** will serve as the **fitting back-end** for the Cardiac Electrophysiology Web Lab

# Problem statement



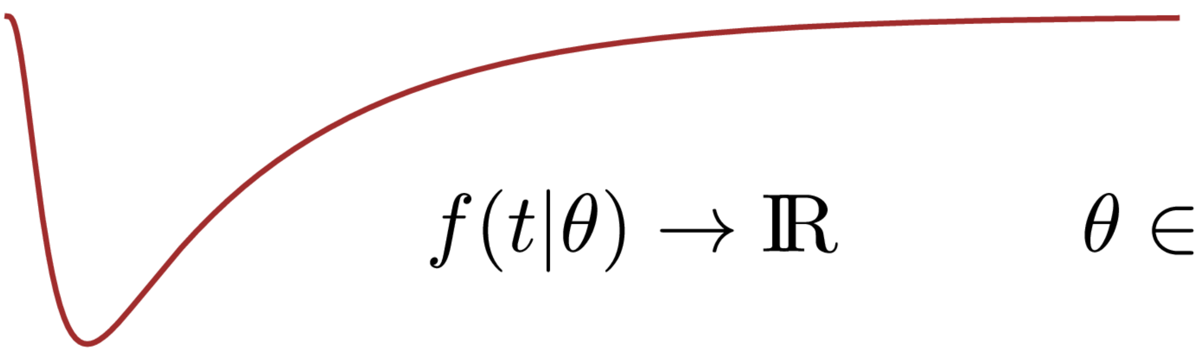
# Problem statement



$$D = \{(t_1, z_1), ..., (t_n, z_n)\}$$

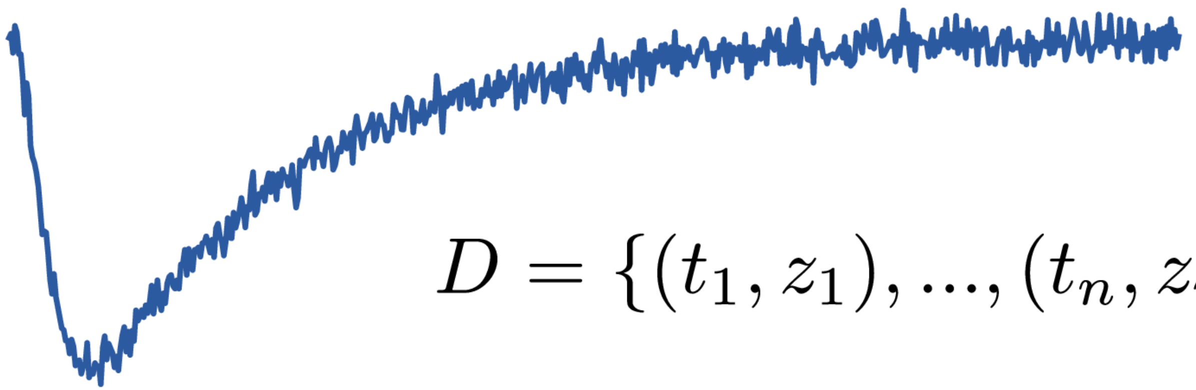
$$t_i > t_{i-1}$$

$$z_i \in \mathbb{R}$$



$$f(t|\theta) \rightarrow \mathbb{R} \qquad \theta \in \mathbb{R}^d$$

# Problem statement



$$D = \{(t_1, z_1), ..., (t_n, z_n)\}$$

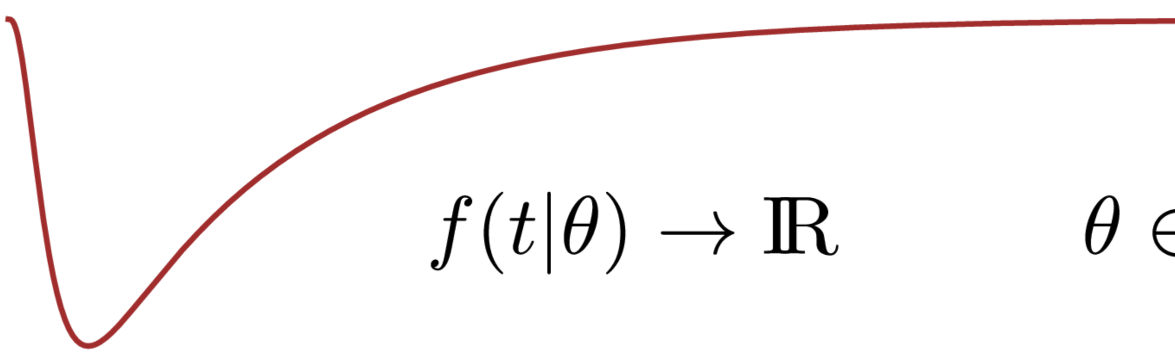
$$t_i > t_{i-1}$$

$$z_i \in \mathbb{R}$$

or:

$$z_i \in \mathbb{R}^m$$

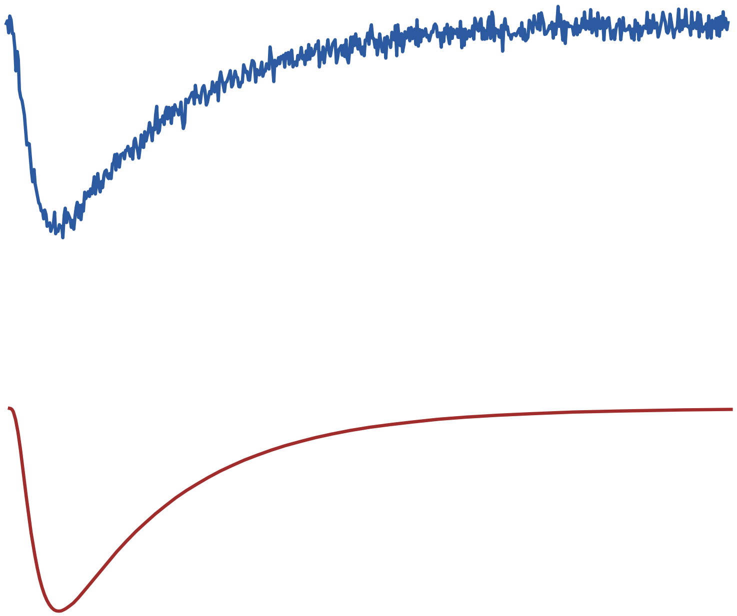
$$f(t|\theta) \rightarrow \mathbb{R}^m$$



$$f(t|\theta) \rightarrow \mathbb{R} \qquad \theta \in \mathbb{R}^d$$

# Problem statement

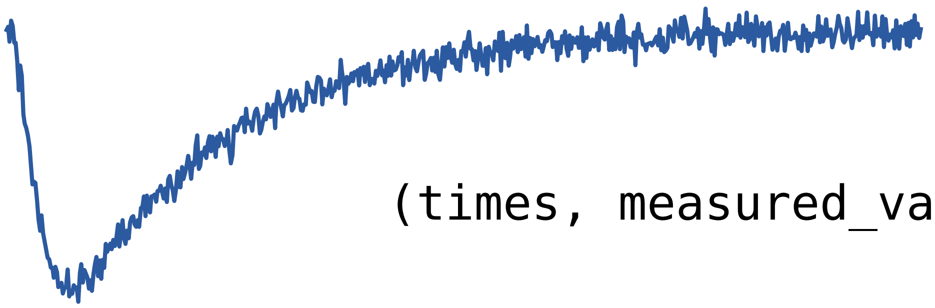
- Given **noisy** experimental **time series**
- And a **forward model** with ***d*** **parameters** that can **predict values** for a given set of **times,** we want to:
  - Find the best set of parameters
  - Check how good they are



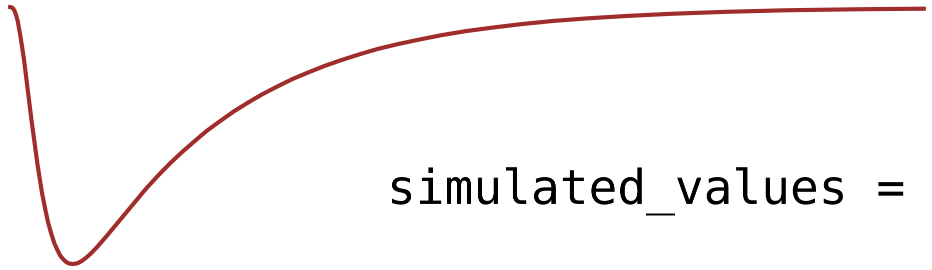
# Methods

- Optimisation
  - Find single best fit
  - Non-linear & derivative-free
- Sampling
  - Find a distribution of probable parameters

# Implementation in Python



`(times, measured_values)`



```
simulated_values = model.simulate(parameters, times)

              d = model.n_parameters()
```

# Implementation in Python

```
class MyModel(pints.Model):
    def n_parameters(self):
        ...
    def simulate(self, parameters, times):
        ...
        # This is where you:
        #  - Write a simple method in Python
        #  - Call your own super C/C++ code
        #  - Call on Chaste, OpenCOR, Myokit, anything
        #  - As long as you can
        return simulated_values
```

# Implementation in Python

```
class MyModel(pints.Model):
    def n_parameters(self):
        ...
    def simulate(self, parameters, times):
        ...
        # This is where you:
        #  - Write a simple method in Python
        #  - Call your own super C/C++ code
        #  - Call on Chaste, OpenCOR, Myokit, anything
        #  - As long as you can
        return simulated_values

problem = SingleSeriesProblem(model, times, measured_values)
```

# Implementation in Python

```
class MyModel(pints.Model):
    def n_parameters(self):
        ...
    def simulate(self, parameters, times):
        ...
        # This is where you:
        #  - Write a simple method in Python
        #  - Call your own super C/C++ code
        #  - Call on Chaste, OpenCOR, Myokit, anything
        #  - As long as you can
        return simulated_values

problem = SingleSeriesProblem(model, times, measured_values)

simulated_values = problem.evaluate(parameters)
measured_values = problem.measured_values()
```

# Implementation in Python

```
class MyModel(pints.Model):
    def n_parameters(self):
        ...
    def simulate(self, parameters, times):
        ...
        # This is where you:
        #  - Write a simple method in Python
        #  - Call your own super C/C++ code
        #  - Call on Chaste, OpenCOR, Myokit, anything
        #  - As long as you can
        return simulated_values

problem = SingleSeriesProblem(model, times, measured_values)

error_measure = SumOfSquares(problem)
```

# Implementation in Python

```
class MyModel(pints.Model):
    def n_parameters(self):
        ...
    def simulate(self, parameters, times):
        ...
        # This is where you:
        #  - Write a simple method in Python
        #  - Call your own super C/C++ code
        #  - Call on Chaste, OpenCOR, Myokit, anything
        #  - As long as you can
        return simulated_values

problem = SingleSeriesProblem(model, times, measured_values)
error_measure = SumOfSquares(problem)

guessed_parameters = [1, 2, 3]
f = error_measure(guessed_parameters)
```

# Implementation in Python

```
class MyModel(pints.Model):
    def n_parameters(self):
        ...
    def simulate(self, parameters, times):
        ...
        # This is where you:
        #  - Write a simple method in Python
        #  - Call your own super C/C++ code
        #  - Call on Chaste, OpenCOR, Myokit, anything
        #  - As long as you can
        return simulated_values

problem = SingleSeriesProblem(model, times, measured_values)
error_measure = SumOfSquares(problem)

initial_point = [1, 2, 3]
optimisation = Optimisation(
   error_measure, initial_point, method=pints.XNES)
```

# Implementation in Python

```python
class MyModel(pints.Model):
    def n_parameters(self):
        ...
    def simulate(self, parameters, times):
        ...
        # This is where you:
        #  - Write a simple method in Python
        #  - Call your own super C/C++ code
        #  - Call on Chaste, OpenCOR, Myokit, anything
        #  - As long as you can
        return simulated_values

problem = SingleSeriesProblem(model, times, measured_values)
error_measure = SumOfSquares(problem)

initial_point = [1, 2, 3]
optimisation = Optimisation(
   error_measure, initial_point, method=pints.XNES)

best_parameters = optimisation.run()
```

# Currently available optimisers

- Natural evolution strategies:
  - CMAES (Hansen et al., 2006)
  - XNES (Glasmachers et al., 2010)
  - SNES (Schaul et al., 2011)
- Particle-based methods
  - PSO (Kennedy & Eberhart, 1995)

# MCMC Sampling in Pints

```
class MyModel(pints.Model):
    def n_parameters(self):
        ...
    def simulate(self, parameters, times):
        ...
        # This is where you:
        #  - Write a simple method in Python
        #  - Call your own super C/C++ code
        #  - Call on Chaste
        #  - As long as you can
        return simulated_values

problem = SingleSeriesProblem(model, times, measured_values)

error_measure = SumOfSquares(problem)
log_likelihood = UnknownNoiseLogLikelihood(problem)

optimisation = Optimisation(error_measure, initial_point)
mcmc = MCMCSampling(log_likelihood, n_chains, initial_points)

best_parameters = optimisation.run()
chains = mcmc.run()
```

# Currently available inference methods:

- Monte Carlo Markov Chain (MCMC):
  - AdaptiveCovarianceMCMC (Haario et al. 2001)
  - DifferentialEvolutionMCMC (Ter Braak et al. 2006)
  - MetropolisRandomWalkMCMC (Metropolis et al. 1953)
  - PopulationMCMC (Jasra et al. 2007)
- Nested sampling
  - NestedEllipsoidSampler (Mukherjee et al. 2008)
  - NestedRejectionSampler (Skilling et al. 2006)

And we're still adding more!

# Boundaries & priors

- All optimisers accept **boundaries:**

```
problem = SingleSeriesProblem(model, times, measured_values)

error_measure = SumOfSquares(problem)

initial = [1, 2, 3]

boundaries = ([0, 0, 0], [5, 5, 5])

optimisation = Optimisation(
   error_measure, initial, boundaries=boundaries, method=CMAES)

best_parameters = optimisation.run()
```

# Boundaries & priors

- **Priors** can be used in sampling:

```
problem = SingleSeriesProblem(model, times, measured_values)

log_likelihood = UnknownNoiseLogLikelihood(problem)

log_prior = UniformLogPrior([0, 0, 0, 1e-5], [5, 5, 5, 1e-3])
log_posterior = LogPosterior(log_likelihood, log_prior)

initial_points = [
   [1, 2, 3, 1e-4], [2, 3, 4, 1e-4], [3, 1, 3, 1e-4]]

mcmc = MCMCSampling(log_posterior, 3, initial_points)

chains = mcmc.run()
```

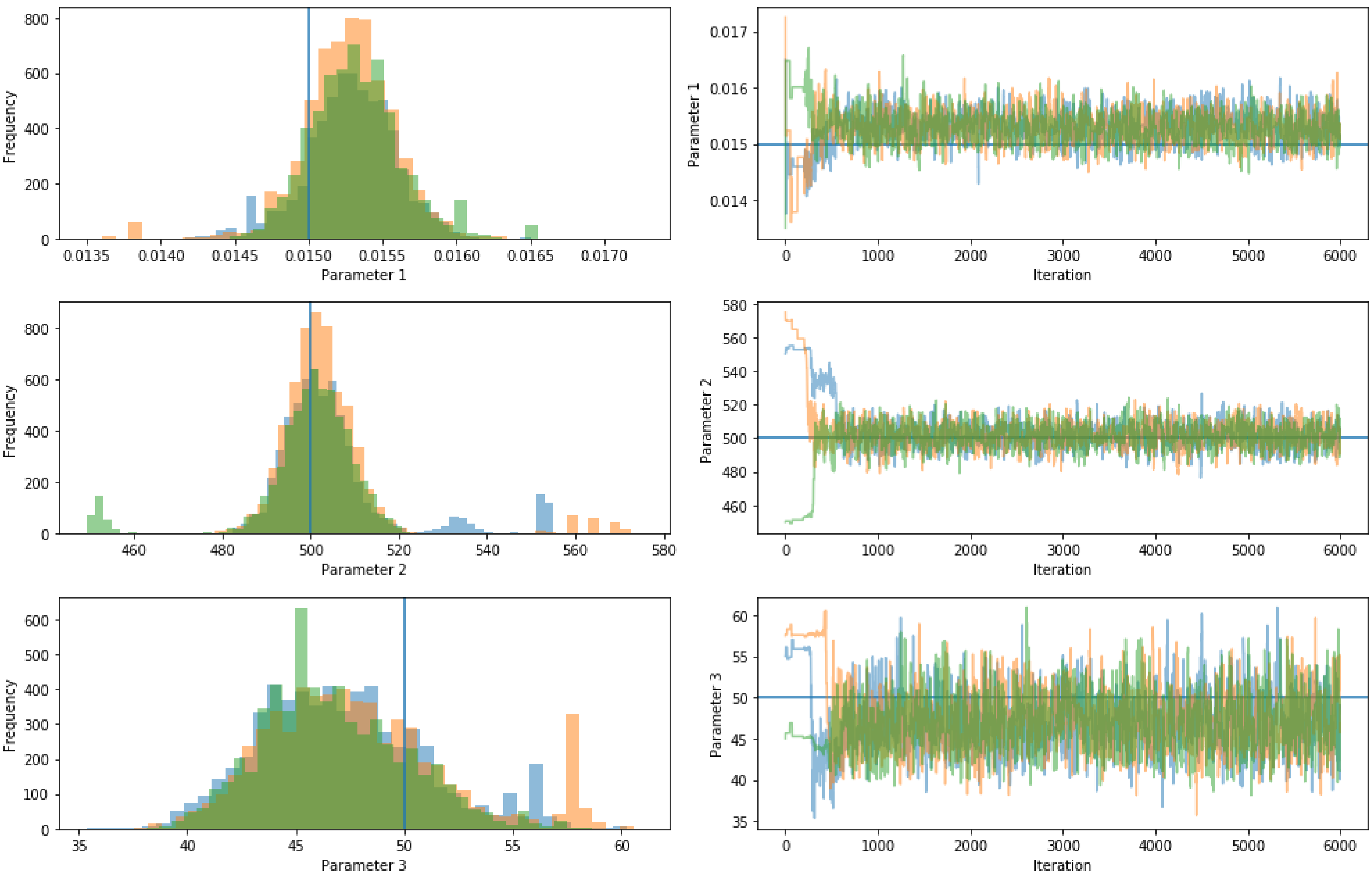
# Ask-and-tell interface

- Most samplers and optimisers use an ask-and-tell interface:

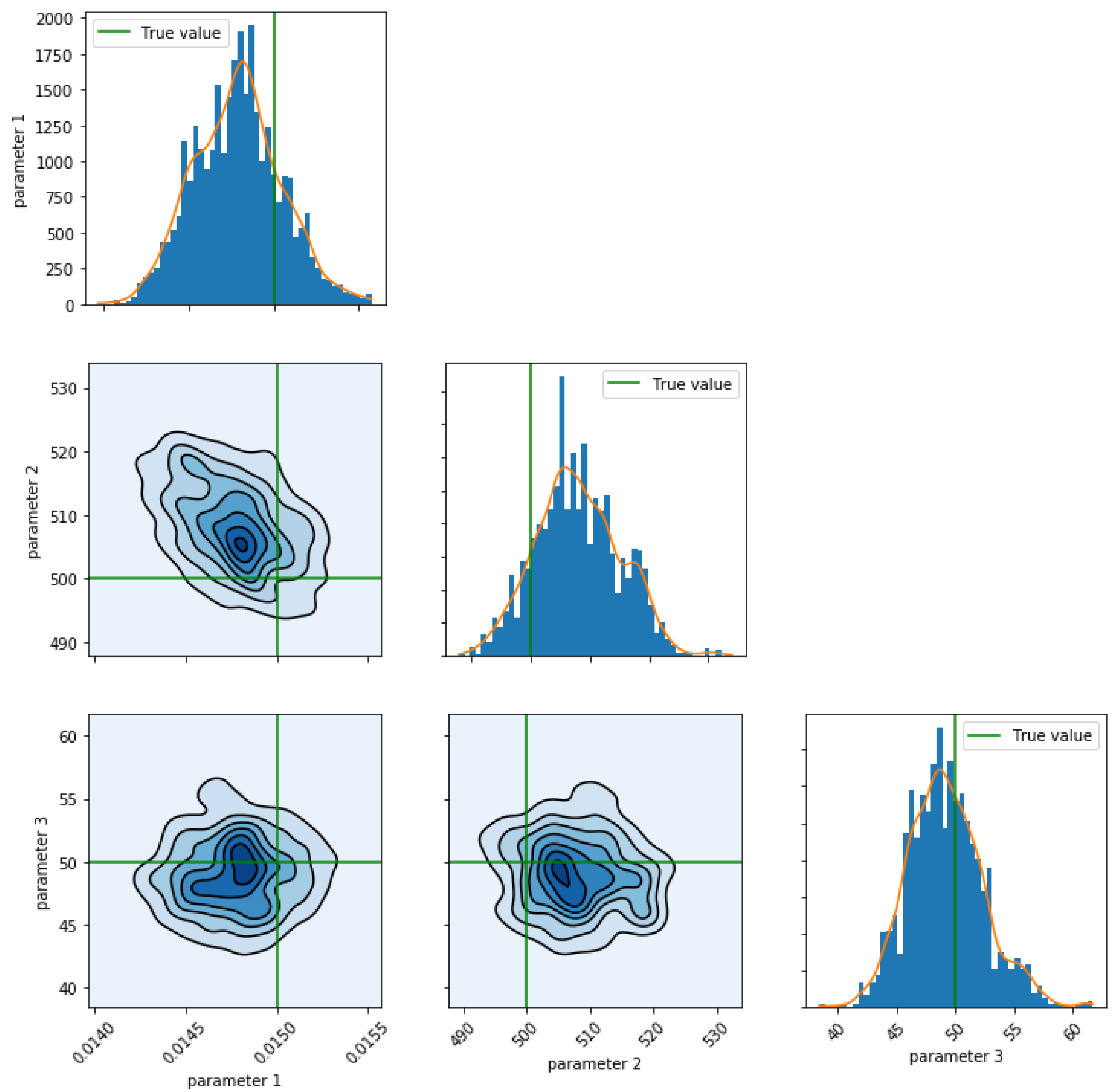
```
next_points = optimiser.ask()

scores = MASSIVE_CLUSTER_ON_THE_SOUTH_POLE.calculate()

optimiser.tell(scores)
```

- This allows fine-grained control, lets users parallelise their simulations, and removes the monolithic “start-and-wait” paradigm

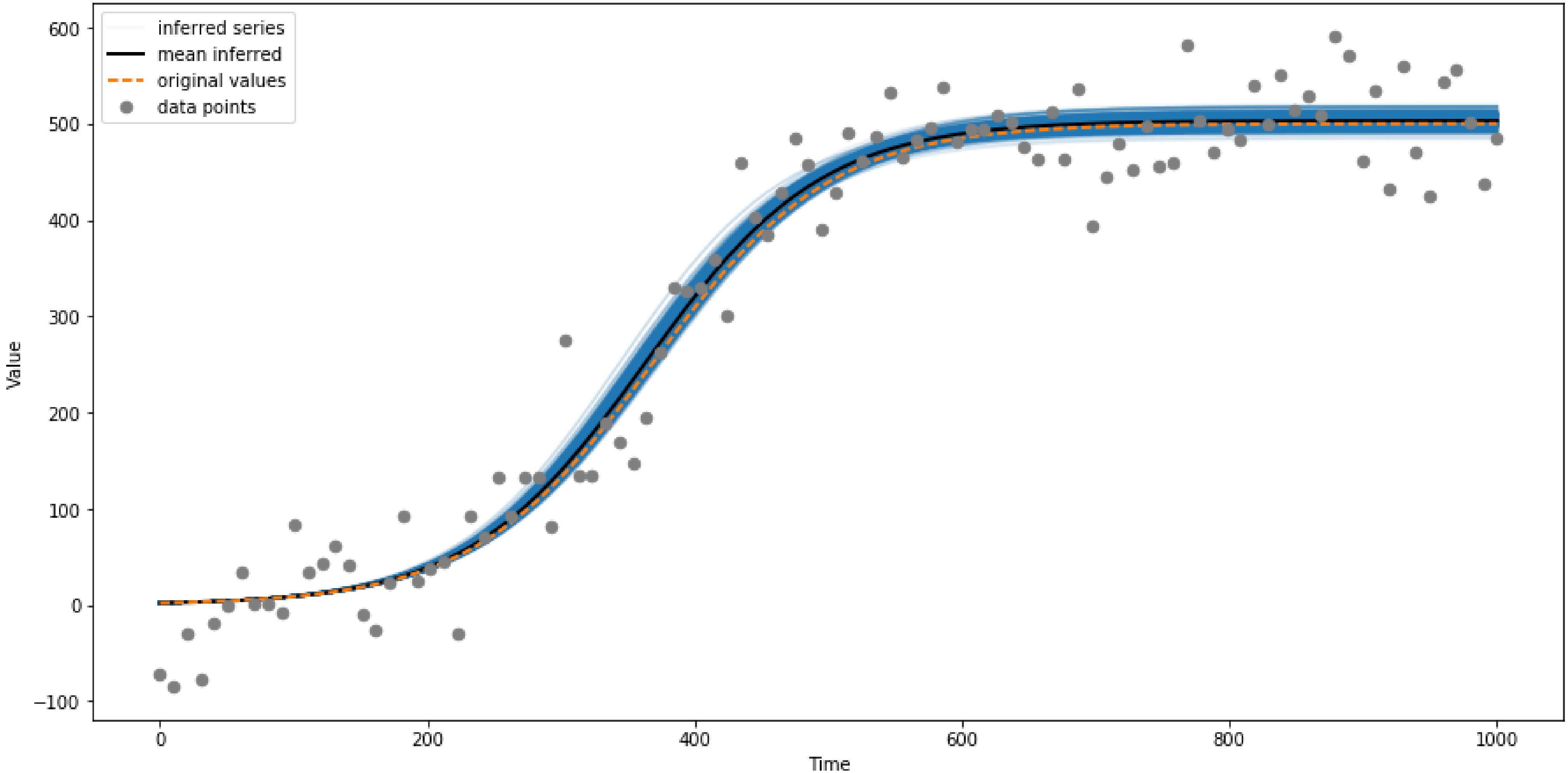
# Diagnostic plots: trace and histogram



# Diagnostic plots: pairwise scatterplots



# Diagnostic plots: predicted time series



# Lots of Jupyter Notebooks on Github!

# Examples

This page contains a number of examples showing how to use Pints.

Each example was created as a *Jupyter notebook* (http://jupyter.org/). These notebooks can be downloaded and used, or you can simply copy/paste the relevant code.

## Getting started

- Optimisation: First example
- Sampling: First example
- Writing a model
- Writing a custom LogPDF
- Writing a custom LogPrior

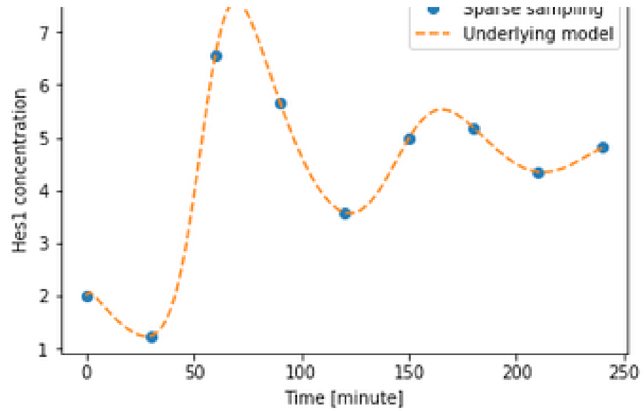
## Optimisation

### Particle-based methods

- CMA-ES
- PSO
- SNES
- XNES

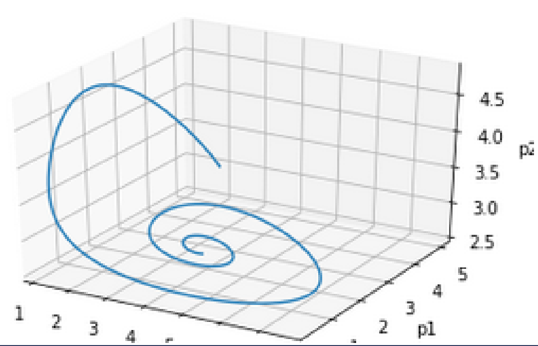
### Further optimisation

# Featuring many real-life examples!




In this set-up, we only see one of the state variables representing the Hes1 concentration. However, it is also interesting to see the dynamics of the whole system (i.e. by inspecting all state variables). This can be done by `simulate_all_states(parameters, times)` provided in this model.

In [3]:
```
all_states = model.simulate_all_states(parameters, smooth_times)

from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.set_xlabel('m')
ax.set_ylabel('p1')
ax.set_zlabel('p2')
plt.plot(all_states[:, 0], all_states[:, 1], all_states[:, 2])
plt.show()
```



# Infrastructure, docs & testing

- Pints is fully open source (BSD 3-clause license)
- 100% Python (2 and 3 compatible)
  - PIP Installable
- Full API docs on http://pints.readthedocs.io
- Continuous integration using TRAVIS
  - Coverage testing using Codecov.io
- Static Jupyter examples on Github
  - Live Jupyter examples using Binder

# Future work

- Add local optimisation methods
  - Via wrappers around e.g. scipy
- Add more sampling methods
  - Including ones using first-order sensitivities
- Add functional/statistical testing
  - In addition to current unit tests
  - Based on famous (hard) “toy” problems

# Thank you!

We invite you all to **use Pints!**
**and contribute methods & problems!**

**https://github.com/pints-team/pints**