Journal of
open research software

Clerx, M, et al. 2019 Probabilistic Inference on Noisy Time Series (PINTS). *Journal of Open Research Software,* 7: 23. DOI: https://doi.org/10.5334/jors.252

SOFTWARE METAPAPER

Probabilistic Inference on Noisy Time Series (PINTS)

Michael Clerx[1], Martin Robinson[1], Ben Lambert[2], Chon Lok Lei[1], Sanmitra Ghosh[1], Gary R. Mirams[3] and David J. Gavaghan[1]

[1] Department of Computer Science, University of Oxford, Oxford, UK

[2] MRC Centre for Global Infectious Disease Analysis, School of Public Health, Imperial College London, UK

[3] Centre for Mathematical Medicine and Biology, School of Mathematical Sciences, University of Nottingham, Nottingham, UK

Corresponding author: Michael Clerx (michael.clerx@cs.ox.ac.uk)

Time series models are ubiquitous in science, arising in any situation where researchers seek to understand how a system's behaviour changes over time. A key problem in time series modelling is *inference*; determining properties of the underlying system based on observed time series. For both statistical and mechanistic models, inference involves finding parameter values, or distributions of parameters values, which produce outputs consistent with observations. A wide variety of inference techniques are available and different approaches are suitable for different classes of problems. This variety presents a challenge for researchers, who may not have the resources or expertise to implement and experiment with these methods. PINTS (Probabilistic Inference on Noisy Time Series — https://github.com/pints-team/pints) is an open-source (BSD 3-clause license) Python library that provides researchers with a broad suite of non-linear optimisation and sampling methods. It allows users to wrap a model and data in a transparent and straightforward interface, which can then be used with custom or pre-defined error measures for optimisation, or with likelihood functions for Bayesian inference or maximum-likelihood estimation. Derivative-free optimisation algorithms — which work without harder-to-obtain gradient information — are included, as well as inference algorithms such as adaptive Markov chain Monte Carlo and nested sampling, which estimate distributions over parameter values. By making these statistical techniques available in an open and easy-to-use framework, PINTS brings the power of these modern methods to a wider scientific audience.

Keywords: Time series models; non-linear optimisation; MCMC sampling; nested sampling; Bayesian inference; Python

Funding statement: M.C., G.R.M. and D.J.G. acknowledge support from the UK Biotechnology and Biological Sciences Research Council [BBSRC grant number BB/P010008/1]; M.R., S.G. and D.J.G. gratefully acknowledge research support from the UK Engineering and Physical Sciences Research Council Cross-Disciplinary Interface Programme [EPSRC grant number EP/I017909/1]; C.L.L. acknowledges support from the Clarendon Scholarship Fund, the EPSRC and the UK Medical Research Council (MRC) [EPSRC grant number EP/L016044/1]; B.L. acknowledges support from the UK Engineering and Physical Sciences Research Council [EPSRC grant number EP/F500394/1]; and S.G. and G.R.M. acknowledge support from the Wellcome Trust & Royal Society [Wellcome Trust grant numbers 101222/Z/13/Z and 212203/Z/18/Z].

(1) Overview

Introduction

Time series models are common in science, where they are used to describe the dynamics of system behaviours. In many cases, these models are non-linear and impossible to solve analytically, so that the *forward problem* (predicting the model output for a given set of parameters) is computationally hard. For such models, there is no single method which can reliably solve the *inverse problem* of estimating parameter values from a noisy time trace. Much like there is a variety of forward models, there is a diversity of approaches for parameter inference. Further, it is often unclear which approach to apply when, meaning that researchers are required to implement a range of methods before successfully fitting their model to data.

PINTS is a software framework that allows users to easily trial and apply different inference methods to their problem. The inference methods supplied by PINTS fall into two broad categories: *optimisers*, which attempt to find a single best parameter vector, and *samplers*, which aim to estimate a probability distribution over parameter values that are compatible with observed results. Users are expected to already have a forward model (for example, a simulation) at their disposal, which they make available to PINTS by writing a simple Python wrapper. They then define a `Problem` (a forward model plus a data set), on which either an `ErrorMeasure` (for optimisation) or a `LogPDF` (for optimisation and sampling) is defined. Currently available optimisers include CMA-ES [6], XNES [5], SNES [21], and Particle Swarm Optimisation (PSO) [12]. Sampling methods include Random Walk Metropolis Markov chain Monte Carlo (MCMC) [13, 15], adaptive covariance MCMC [10], Population MCMC [8], Differential Evolution [23], DREAM [25], emcee [3], Hamiltonian MCMC [17], and MALA [4]. In addition, ellipsoidal [16] and rejection nested samplers [22] are provided. Convenience plotting methods are provided to quickly visualise the results, as well as diagnostic tools to inspect the validity of the results. An example of an optimisation problem and its solution using PINTS is shown in **Figure 1**.

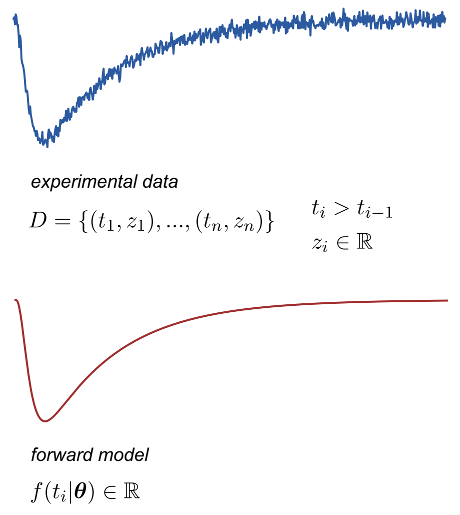
PINTS was developed as a community effort by researchers in electrochemistry, cardiac electrophysiology, and statistics, to compare different methods for solving inverse problems in a common framework. It features a clean and transparent object-orientated API that is designed to easily accommodate new error measures, log-likelihoods, optimisers and samplers, allowing users to utilise pre-built components as much as possible, while adding their own code for problem-specific areas. The PINTS team aims for full test coverage, and includes unit testing and extended statistical tests to verify the correct operation of all methods.

Early research using PINTS in electrochemistry has included fitting a differential-algebraic equation (DAE) model of reduction-oxidation to voltammetry measurements of a Polyoxometalates molecule [19], and the design and application of a custom hierarchical statistical model for repeat voltammetry experiments of a Ferricynide process [18].

Optimisation algorithms are implemented in many different software packages, (see, for example, the Python `scipy.optimize` module), but are often biased towards gradient-based methods, which can perform poorly for many ordinary and partial differential equations used in time series modelling. PINTS therefore focuses on derivative-free optimisers, although we plan to add gradient-based methods for comparison. In contrast with more general-purpose optimisation software, PINTS contains a number of error measures specifically suited to time series models, and adds the ability to use any PINTS log-likelihood class as an error measure in order to perform maximum likelihood estimation (MLE).

Dakota [1] is a widely regarded package for parameter fitting and uncertainty quantification and is most similar to PINTS in that it offers a generic interface to call an (assumed expensive) model, as well as a wide variety of optimisers and samplers. In contrast to the PINTS Python API, Dakota uses either a C++ or file input/output process for communication between user models and the library, and does not provide options for specifying either the error measures or the log-likelihoods of the inverse problem. However, Dakota has some features not yet available in PINTS, such as the option to train a surrogate model, useful for very expensive model evaluations.

Other software packages that enable parameter inference and sampling for ODE models include BioBayes [26], ABC-SysBio [14], SYSBIONS [9], PyMC3 [20], and



```
import pints

class MyModel(pints.ForwardModel):
    def n_parameters(self):
        return 3
    def simulate(self, parameters, times):
        ... Run a simulation with the given parameters and times
        return simulated_values

times, values = ... Load the experimental times and values

problem = pints.SingleOutputProblem(
    MyModel(), times, values)

error_measure = pints.SumOfSquaresError(problem)

initial_point = [1, 2, 3]

optimisation = pints.OptimisationController(
    error_measure, initial_point, method=pints.XNES)

parameters, error = optimisation.run()
```

Figure 1: *(Left)* An experimentally measured noisy time series, and a simulated one. *(Right)* An example of an optimisation procedure with PINTS. Note that the actual simulation code is omitted from the example model wrapper at the top: this is the user-provided part, and can be written in Python or any other language that can interface with Python, allowing computationally heavy forward simulations to be handled entirely outside of PINTS. This image, and the full example code, can also be found in the PINTS repository.

Stan [2]. These packages use either a common model description format (for example, SBML) or their own language (for example, Stan's probabilistic programming language) to specify the model, presenting additional learning hurdles for a user and often restricting the class of models which can be fit. By contrast, PINTS aims to be as general as possible to support a wider variety of models (for example, PDEs). PyMC3 [20] does provide a similar generic model interface to PINTS but, as with the other packages, specialises in one sampling method, whereas PINTS aims to support a wide variety of methods with the assumption that no one sampling method is suitable for all models of interest. BCM [24] offers both a generic interface (via C++) and a wide variety of samplers, but does not supply any likelihood functions and is unfortunately largely undocumented.

Implementation and architecture

PINTS is designed around two core ideas: 1. PINTS should work with a wide range of time series models, and make no demands on how they are implemented other than a minimal input/output interface. 2. It is assumed that model evaluation (simulation) is the most costly step in any optimisation or sampling routine.

The decision to use Python fits both these criteria: Python interfaces well with C and C++, which are typically used for high-performance simulation, and any performance hit of using the high-level, easy to read and write language Python is overshadowed by simulation time.

Defining an optimisation or sampling problem

All optimisers operate on a callable `ErrorMeasure` object that describes a function to minimise or on a callable `LogPDF` object that describes a probability density function (PDF) to maximise. Similarly, all samplers start from a callable `LogPDF`, so that the same probability function can be used with both optimisers and samplers. The natural logarithm of the PDF is used for computational efficiency and accuracy, and we allow the probability density to be unnormalised (i.e. its integral does not have to sum to 1). **Figure 2** shows how a user-defined model can be wrapped in a PINTS `ForwardModel` and combined with time points and measured values to create a `Problem` from which several standard `ErrorMeasure`s and `LogPDF`s can be created. For inference in a Bayesian context, a `LogPosterior` class and several `LogPrior` distributions are provided. If a given `LogPDF` or `ErrorMeasure` cannot be constructed from PINTS classes, users can also define their own classes.

Implementation of optimisers and samplers

Most PINTS samplers and optimisers are implemented using a so-called *ask-and-tell interface*, inspired by the Python implementation of CMA-ES [6] (https://github.com/CMA-ES/pycma). In this framework, the details of solving the forward problem are partitioned away from the rest of the sampling or optimising algorithm. For each iteration the following steps are undertaken (**Figure 3**): first, the user calls `ask()` to obtain one or more parameter values from their chosen method — these values are typically vectors generated stochastically conditional on an internal system state; second, the user solves the forward model and generates a score for each parameter vector, for example, an error measure or (unnormalised) posterior probability; third, the user calls `tell()` to pass the score back to the method, which can then update its internal state and finish the iteration. For example, in

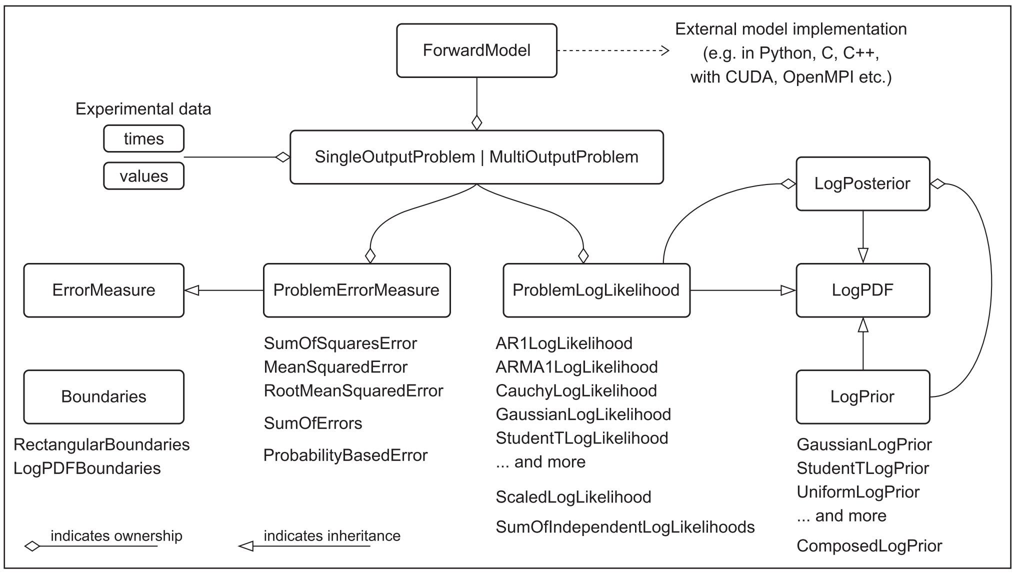


Figure 2: An overview of the main PINTS classes used to define error measures (for optimisation) and PDFs (for optimisation or sampling). Users write a wrapper class for their model, making it available to pints, and must provide the experimental data using any Python sequence structure (for example, a list or a `NumPy` array). With these ingredients, a (single or multi-output) problem can be defined that can then be used with any of the available error measure or likelihood classes. Alternatively, users implement their own `ErrorMeasure` or `LogPDF`, which allows for further customisation and for problems other than time series problems to be solved.

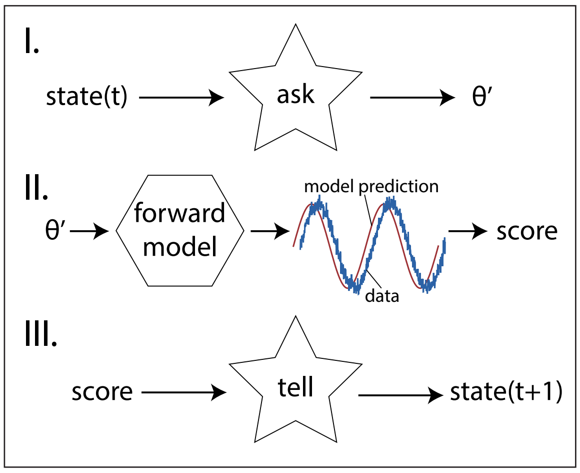


Figure 3: The three steps iterated in an ask-and-tell interface. The stars here represent code specific to the chosen sampling or optimiser method and θ' is the input parameter vector proposed by `ask()`. The `state(.)` of the system varies according to the method but typically holds a set of input parameter vectors and other constant or dynamic variables used by the `ask()` and `tell()` steps.

many MCMC methods each `ask()` call returns a single proposed sample to be evaluated by the user, and the following `tell()` then either accepts or rejects this point based on its probability. For optimisers such as CMA-ES, `ask()` returns a set of points in parameter space, and the scores passed in via `tell()` are then used to estimate the local gradient, which is used to move towards the estimated optimum. This framework has a number of advantages: since optimisation or sampling can take hours or even days, this allows programs using PINTS to provide regular user feedback and logging (which is not possible when the routine is implemented as a single monolithic function call); allows users with access to CPU clusters or GPU machines to implement their own parallelised evaluation of `ErrorMeasures` and `LogPDF`s; lets users implement their own strategies (for example, by dynamically changing hyperparameters) and/or stopping criteria; and, finally, by delineating the sampling or optimisation algorithm's steps from the methods used to solve the forward problem, encourages development of transparent and modular code.

Running optimisation and sampling

For more casual users (whom we expect will be the majority), PINTS includes 'controller' classes (e.g. `OptimisationController` or `MCMCController`) that provide a higher level interface (see e.g. **Figure 1**). When creating a controller, the user specifies the name of the lower-level method to use (e.g. `CMAES`, or `AdaptiveCovarianceMCMC`) and passes in an `ErrorMeasure` or `LogPDF`. The controller then instantiates and manages this method, as well as providing user-configurable stopping criteria (e.g. maximum number of iterations) and logging to screen and/or the filesystem. An important difference with the lower-level ask-and-tell interface, is that controllers handle function evaluation, and can distribute evaluations over multiple CPU cores using Python multiprocessing. This provides users with out-of-the-box parallelisation, while users looking to implement custom parallelisation can fall back on the ask-and-tell interface.

Quality control

PINTS has three levels of testing: unit testing, functional testing, and comparative testing. Unit tests are used to test the functionality of simple (deterministic) methods, and to check that complex (pseudo-random) methods run without raising exceptions. All the unit tests are available to be run by a user to ensure the software is working correctly. Continuous integration is carried out using Travis CI (Ubuntu Trusty distribution with Python versions 2.7, 3.4, 3.5 and 3.6, and OS/X with Python version 2.7) and AppVeyor (Windows Server 2011 R2 with Python versions 2.7, 3.4, 3.5 and 3.6). PINTS uses Flake8 linter tests to ensure that contributed code conforms to best practices and code coverage tests to ensure that all code is sufficiently tested and documented. Functional testing of the methods is performed separately and is used to test the method's behaviour from different (pseudo-random) initial conditions. Analysis of functional checking is done both visually and statistically (for example, if recent results deviate significantly from previous results this indicates the possible introduction of a bug). In our current solution, functional checking is run automatically but the results are interpreted by the PINTS developers. Work is ongoing to develop an automated analysis system so that these functional tests can be included in the continuous integration pipeline. Finally, in comparative testing a number of problems are set up and solved with different methods in order to compare the solutions they return and evaluate their performance. Performance testing is partially automated, but tests are initiated and analysed by the developer.

(2) Availability

Operating system

PINTS uses no functions specific to any operating system (OS) and so can run on any OS that provides Python. Optional parallelisation is provided that uses the Python `multiprocessing` module, which works best on UNIX-based systems (for example, Linux and OS/X), but runs on Windows with slightly reduced performance.

Programming language

PINTS requires Python 2.7 or higher, or Python 3.4 or higher.

Additional system requirements

PINTS has a minimal disk space footprint (approximately 2MB) and can be run on single-processor devices or headless on multi-processor machines (for example, via `ssh`).

Dependencies

PINTS uses the NumPy (version 1.8 or higher) and SciPy (version 0.14 or higher, [11]) libraries extensively. The default optimisation method is CMA-ES, for which the `cma` package (version 2 or higher) is used. The remaining optimisation and sampling methods require no further dependencies. Finally, Matplotlib (version 1.5 or higher [7]) is required for installation, but it is possible to use PINTS without Matplotlib or with a different plotting library.

List of contributors

Michael Clerx, Sanmitra Ghosh, Ben Lambert, Chon Lok Lei, Martin Robinson.

Software location

Name: GitHub (release v0.2.2)
Persistent identifier: https://github.com/pints-team/pints/releases/tag/v0.2.2
Licence: BSD 3-clause
Publisher: Pints team
Version published: 0.2.2
Date published: 30/04/2019

Code repository

Name: GitHub (develop)
Persistent identifier: https://github.com/pints-team/pints
Licence: BSD 3-clause
Date published: 16/05/17

Language

English.

(3) Reuse potential

Detailed documentation is provided on using PINTS with user-supplied models (see, for example, the `writing-a-model` example on the GitHub repository). While PINTS was designed primarily with biological and electrochemical problems in mind, there is nothing to prohibit its use on time series models from other fields. Similarly, while the implemented `ErrorMeasure` and `LogPDF` classes were chosen to work well with time series problems, PINTS can be used outside this setting (in fact, utility functions `fmin` and `curve_fit` are provided specifically for this purpose). Users are also free to create their own `ErrorMeasure` or `LogPDF` classes that do not rely on PINTS `ForwardModel` or `Problem` classes, which allows PINTS to be used beyond its intended scope of time-series problems. A link to the full API documentation can be found on the GitHub repository, which also contains a list of examples for all PINTS' main features (https://github.com/pints-team/pints/tree/master/examples). We have found that these examples, rather than our API documentation, serve to kick-start any new project based on PINTS.

Contact and support

We welcome questions, suggestions, bug reports, and user contributions via the GitHub repository, which acts as a central communication platform for PINTS. A detailed guide on contributing to PINTS is also available there. A Research Software Engineering group has recently been established at Oxford university, through which we hope to provide long-term support. In addition, we are happy to respond to questions via e-mail or other means of communication.

Competing Interests

The authors have no competing interests to declare.

Author Contributions

Michael Clerx and Martin Robinson authors contributed equally to this manuscript.

References

1. **Adam, B, Bauman, L, Bohnhoff, W, Dalbey, K, Ebeida, M,** et al. 2015 'Dakota, a multilevel parallel object-oriented framework for design optimization, parameter estimation, uncertainty quantification, and sensitivity analysis: Version 6.0 user manual', Technical report, Tech. rep., Sandia NationalLaboratories. DOI: https://doi.org/10.2172/1177048
2. **Carpenter, B, Gelman, A, Hoffman, M D, Lee, D, Goodrich, B, Betancourt, M, Brubaker, M, Guo, J, Li, P** and **Riddell, A** 2017 'Stan: A probabilistic programming language'. *Journal of Statistical Software*, 76(1). DOI: https://doi.org/10.18637/jss.v076.i01
3. **Foreman-Mackey, D, Hogg, D W, Lang, D** and **Goodman, J** 2013 'emcee: the MCMC hammer'. *Publications of the Astronomical Society of the Pacific*, 125(925): 306. DOI: https://doi.org/10.1086/670067
4. **Girolami, M** and **Calderhead, B** 2011 'Riemann manifold Langevin and Hamiltonian Monte Carlo methods'. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 73(2): 123–214. DOI: https://doi.org/10.1111/j.1467-9868.2010.00765.x
5. **Glasmachers, T, Schaul, T, Yi, S, Wierstra, D** and **Schmidhuber, J** 2010 'Exponential natural evolution strategies'. In: '*Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation', ACM*, 393–400. DOI: https://doi.org/10.1145/1830483.1830557
6. **Hansen, N, Müller, S D** and **Koumoutsakos, P** 2003 'Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (CMA-ES)'. *Evolutionary Computation*, 11(1): 1–18. DOI: https://doi.org/10.1162/106365603321828970
7. **Hunter, J D** 2007 'Matplotlib: A 2D graphics environment'. *Computing in Science & Engineering*, 9(3): 90–95. DOI: https://doi.org/10.1109/MCSE.2007.55
8. **Jasra, A, Stephens, D A** and **Holmes, C C** 2007 'On population-based simulation for static inference'. *Statistics and Computing*, 17(3): 263–279. DOI: https://doi.org/10.1007/s11222-007-9028-9
9. **Johnson, R, Kirk, P** and **Stumpf, M P** 2014 'SYSBIONS: nested sampling for systems biology'. *Bioinformatics*, 31(4): 604–605. DOI: https://doi.org/10.1093/bioinformatics/btu675
10. **Johnstone, R H, Chang, E T, Bardenet, R, De Boer, T P, Gavaghan, D J, Pathmanathan, P, Clayton, R H** and **Mirams, G R** 2016. 'Uncertainty and variability in models of the cardiac action potential: Can we build trustworthy models?' *Journal of Molecular and Cellular Cardiology*, 96: 49–62. DOI: https://doi.org/10.1113/JP271671
11. **Jones, E, Oliphant, T** and **Peterson, P** 2014 'Scipy: open source scientific tools for Python'. http://www.scipy.org. [Online; accessed 13-Aug-2018].

12. **Kennedy, J** 2011 Particle swarm optimization. In: '*Encyclopedia of Machine Learning*', Springer, 760–766. DOI: https://doi.org/10.1007/978-0-387-30164-8_630
13. **Lambert, B** 2018 *'A Student's Guide to Bayesian Statistics'*. Sage Publications Ltd.
14. **Liepe, J, Barnes, C, Cule, E, Erguler, K, Kirk, P, Toni, T** and **Stumpf, M P** 2010 'ABC-SysBio-approximate Bayesian computation in Python with GPU support'. *Bioinformatics*, 26(14): 1797–1799. DOI: https://doi.org/10.1093/bioinformatics/btq278
15. **Metropolis, N, Rosenbluth, A W, Rosenbluth, M N, Teller, A H** and **Teller, E** 1953 'Equation of state calculations by fast computing machines'. *The Journal of Chemical Physics*, 21(6): 1087–1092. DOI: https://doi.org/10.1063/1.1699114
16. **Mukherjee, P, Parkinson, D** and **Liddle, A R** 2006 'A nested sampling algorithm for cosmological model selection'. *The Astrophysical Journal Letters*, 638(2): L51. URL: https://arxiv.org/pdf/astro-ph/0508461. DOI: https://doi.org/10.1086/501068
17. **Neal, R M,** et al. 2011 'MCMC using Hamiltonian dynamics'. *Handbook of markov chain monte carlo*, 2(11): 2. DOI: https://doi.org/10.1201/b10905-6
18. **Robinson, M, Bond, A, Simonov, A, Zhang, J** and **Gavaghan, D** 2018 'Separating the effects of experimental noise from inherent system variability in voltammetry: The $[Fe(CN)_6]^{3/4}$ process'. *ChemRxiv*. DOI: https://doi.org/10.26434/chemrxiv.7149281.v1
19. **Robinson, M, Ounnunkad, K, Zhang, J, Gavaghan, D** and **Bond, A** 2018 'Integration of heuristic and automated parametrization of three unresolved twoelectron surfaceconfined polyoxometalate reduction processes by AC voltammetry'. *ChemElectroChem*. DOI: https://doi.org/10.1002/celc.201800950
20. **Salvatier, J, Wiecki, T V** and **Fonnesbeck, C** 2016 'Probabilistic programming in Python using PyMC3'. *PeerJ Computer Science*, 2: e55. DOI: https://doi.org/10.7717/peerj-cs.55
21. **Schaul, T, Glasmachers, T** and **Schmidhuber, J** 2011 'High dimensions and heavy tails for natural evolution strategies'. In: *'Proceedings of the 13th annual conference on Genetic and evolutionary computation', ACM*, 845–852. DOI: https://doi.org/10.1145/2001576.2001692
22. **Skilling, J,** et al. 2006 'Nested sampling for general Bayesian computation'. *Bayesian Analysis*, 1(4): 833–859. DOI: https://doi.org/10.1214/06-BA127
23. **Ter Braak, C J** 2006 'A Markov Chain Monte Carlo version of the genetic algorithm Differential Evolution: easy Bayesian computing for real parameter spaces'. *Statistics and Computing*, 16(3): 239–249. DOI: https://doi.org/10.1007/s11222-006-8769-1
24. **Thijssen, B, Dijkstra, T M, Heskes, T** and **Wessels, L F** 2016 'BCM: toolkit for Bayesian analysis of computational models using samplers'. *BMC Systems Biology*, 10(1): 100. DOI: https://doi.org/10.1186/s12918-016-0339-3
25. **Vrugt, J A, Ter Braak, C, Diks, C, Robinson, B A, Hyman, J M** and **Higdon, D** 2009 'Accelerating Markov chain Monte Carlo simulation by differential evolution with self-adaptive randomized subspace sampling', *International Journal of Nonlinear Sciences and Numerical Simulation*, 10(3): 273–290. DOI: https://doi.org/10.1515/IJNSNS.2009.10.3.273
26. **Vyshemirsky, V** and **Girolami, M** 2008 'BioBayes: a software package for Bayesian inference in systems biology'. *Bioinformatics*, 24(17): 1933–1934. DOI: https://doi.org/10.1093/bioinformatics/btn338

How to cite this article: Clerx, M, Robinson, M, Lambert, B, Lei, C L, Ghosh, S, Mirams, G R and Gavaghan, D J 2019 Probabilistic Inference on Noisy Time Series (PINTS). *Journal of Open Research Software,* 7: 23. DOI: https://doi.org/10.5334/jors.252

Submitted: 07 November 2018 **Accepted:** 05 July 2019 **Published:** 19 July 2019