SOFTWARE TOOL ARTICLE

REVISED Reproducible parallel inference and simulation of stochastic state space models using odin, dust, and mcstate [version 2; peer review: 2 approved]

Richard G. FitzJohn[1], Edward S. Knock[1], Lilith K. Whittles[1,2], Pablo N. Perez-Guzman[1], Sangeeta Bhatia[1], Fernando Guntoro[1], Oliver J. Watson[1], Charles Whittaker[1], Neil M. Ferguson[1], Anne Cori[1], Marc Baguelin[1,3], John A. Lees[1]

[1]MRC Centre for Global Infectious Disease Analysis; and the Abdul Latif Jameel Institute for Disease and Emergency Analytics (J-IDEA), School of Public Health, Imperial College London, London, W2 1PG, UK
[2]Modelling and Economics Unit, National Infection Service, Public Health England, London, UK
[3]Department of Infectious Disease Epidemiology, London School of Hygiene & Tropical Medicine, London, WC1E 8HT, UK

v2 **First published:** 11 Dec 2020, **5**:288
https://doi.org/10.12688/wellcomeopenres.16466.1
Latest published: 10 Jun 2021, **5**:288
https://doi.org/10.12688/wellcomeopenres.16466.2

Abstract

State space models, including compartmental models, are used to model physical, biological and social phenomena in a broad range of scientific fields. A common way of representing the underlying processes in these models is as a system of stochastic processes which can be simulated forwards in time. Inference of model parameters based on observed time-series data can then be performed using sequential Monte Carlo techniques. However, using these methods for routine inference problems can be made difficult due to various engineering considerations: allowing model design to change in response to new data and ideas, writing model code which is highly performant, and incorporating all of this with up-to-date statistical techniques. Here, we describe a suite of packages in the R programming language designed to streamline the design and deployment of state space models, targeted at infectious disease modellers but suitable for other domains. Users describe their model in a familiar domain-specific language, which is converted into parallelised C++ code. A fast, parallel, reproducible random number generator is then used to run large numbers of model simulations in an efficient manner. We also provide standard inference and prediction routines, though the model simulator can be used directly if these do not meet the user's needs. These packages provide guarantees on reproducibility and performance, allowing the user to focus on the model itself, rather than the underlying computation. The ability to automatically generate high-performance code that would be tedious and time-consuming to write and verify manually,

Open Peer Review

Reviewer Status ✓ ✓

	Invited Reviewers	
	1	**2**
version 2 (revision) 10 Jun 2021	✓ report	
version 1 11 Dec 2020	? report	✓ report

1. **Edward Ionides**, University of Michigan, Ann Arbor, USA
2. **Rene Niehus**, Harvard T H Chan School of Public Health, Boston, USA

Any reports and responses or comments on the article can be found at the end of the article.

particularly when adding further structure to compartments, is crucial for infectious disease modellers. Our packages have been critical to the development cycle of our ongoing real-time modelling efforts in the COVID-19 pandemic, and have the potential to do the same for models used in a number of different domains.

Keywords
Epidemiology, Infectious diseases, Compartmental models, State space model, Particle filter, SMC, MCMC

Corresponding author: John A. Lees (j.lees@imperial.ac.uk)

Author roles: FitzJohn RG: Conceptualization, Data Curation, Investigation, Methodology, Software, Supervision, Validation, Visualization, Writing – Review & Editing; **Knock ES**: Conceptualization, Data Curation, Investigation, Methodology, Software, Validation, Writing – Review & Editing; **Whittles LK**: Conceptualization, Data Curation, Investigation, Methodology, Software, Validation, Visualization, Writing – Review & Editing; **Perez-Guzman PN**: Data Curation, Investigation, Methodology, Software, Validation, Writing – Review & Editing; **Bhatia S**: Software, Writing – Review & Editing; **Guntoro F**: Software, Writing – Review & Editing; **Watson OJ**: Methodology, Software, Writing – Review & Editing; **Whittaker C**: Software, Writing – Review & Editing; **Ferguson NM**: Funding Acquisition, Methodology, Supervision, Validation, Writing – Review & Editing; **Cori A**: Data Curation, Funding Acquisition, Investigation, Methodology, Software, Supervision, Writing – Review & Editing; **Baguelin M**: Conceptualization, Data Curation, Funding Acquisition, Investigation, Methodology, Software, Supervision, Validation, Writing – Review & Editing; **Lees JA**: Conceptualization, Data Curation, Formal Analysis, Investigation, Methodology, Software, Supervision, Validation, Visualization, Writing – Original Draft Preparation, Writing – Review & Editing

Competing interests: No competing interests were disclosed.

Grant information: JAL was funded by Wellcome [219699]. This work was also supported by the NIHR HPRU in Modelling and Health Economics, a partnership between PHE, Imperial College London and LSHTM (grant code NIHR200908); and acknowledges funding from the MRC Centre for Global Infectious Disease Analysis (reference MR/R015600/1), jointly funded by the UK Medical Research Council (MRC) and the UK Foreign, Commonwealth \& Development Office (FCDO), under the MRC/FCDO Concordat agreement and is also part of the EDCTP2 programme supported by the European Union. The views expressed are those of the authors and not necessarily those of the United Kingdom (UK) Department of Health and Social Care, the National Health Service, the National Institute for Health Research (NIHR), or Public Health England (PHE).
The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

How to cite this article: FitzJohn RG, Knock ES, Whittles LK *et al.* **Reproducible parallel inference and simulation of stochastic state space models using odin, dust, and mcstate [version 2; peer review: 2 approved]** Wellcome Open Research 2021, **5**:288 https://doi.org/10.12688/wellcomeopenres.16466.2

First published: 11 Dec 2020, **5**:288 https://doi.org/10.12688/wellcomeopenres.16466.1

REVISED **Amendments from Version 1**

We have added four new pieces of functionality to the code suggested by the reviewers:

- The iterated filtering algorithm of Ionides *et al.*
- More flexible parallelisation over both chains and particles in pMCMC.
- The ability to add arbitrary C++ code snippets to odin models.
- A simulate function, to more easily run models across a whole time series.

We document these features both in the updated text, and in a number of new package vignettes. In the text, we now include the sections on maximum likelihood inference (rather than entirely operating within a Bayesian framework), and a discussion on how to use these techniques for model criticism. We have also added a section on performing prior predictive checks within our framework. Finally, we have made numerous small changes to improve the clarity of the manuscript, including a new figure which gives an overview of our packages and how they operate together.

Any further responses from the reviewers can be found at the end of the article

Introduction

To mathematically model a physical or biological process one must develop and test a model, combine it with potentially noisy or poor quality data, and then produce high-quality reproducible results in a computationally efficient manner. This constitutes a multi-disciplinary challenge[1,2]. Frameworks which automate common computational and statistical methods can facilitate some of the complex steps in the process, and come with guarantees of efficiency and reproducibility[3,4] – a necessity in modern science, particularly when this science is used in real time to support policy making[5–7].

When designing these frameworks, it is generally fair to assume that a typical multi-disciplinary modeller is a domain expert and technically minded, but should not have to become a software engineer in order to develop an efficient implementation[3]. Therefore, as developers of computational frameworks, we should aim to lower the barriers of entry by using a programming language favoured by researchers in the targeted domain, designing a clear and well-documented application-programmer interface (API), and making installation, use and reuse as painless and portable as possible. We can enhance uptake by combining sensible software engineering choices with carefully designed statistical and computational methods. This make design advantages such as speed, unit-tested code and reproducible random number generation as broadly accessible as possible.

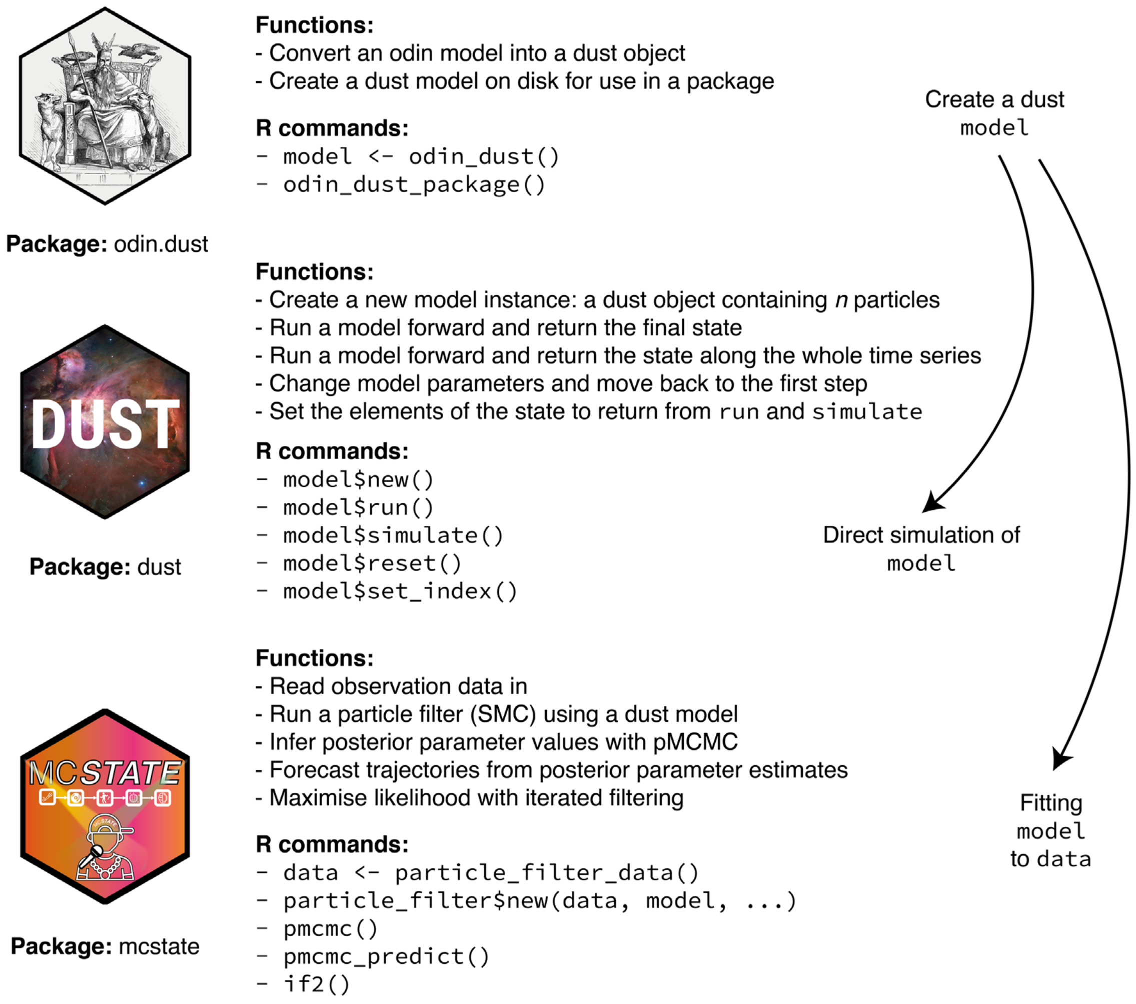
With these aims in mind, we describe the development of three libraries in the R programming language[8], designed to make the implementation of state space models as easy and reliable as possible. R-like code in the odin domain-specific language (DSL) is automatically transpiled (converted between two high-level programming languages) into C++. This C++ code is then compiled into a dynamic library (loaded only when needed) with an R interface. The resulting code is portable, and the generated libraries are lightweight and computationally efficient. This procedure offers the performance and careful memory management of compiled code, without requiring the user to have any specialist programming knowledge. Additionally, we include an R package, mcstate, which provides routines for common inference and prediction tasks. Compiled models also link with functions directly callable from R, so users are free to develop more flexible uses and inference procedures using R programming. An overview of these packages and how they interact is shown in Figure 1. We provided detailed examples of applying these tools to simple stochastic epidemic models, both here and in the package documentation.

As single-threaded code, our packages run at around the same speed to two existing high-performance packages with similar functionality, pomp[9] and libBi[10]. On top of this, we have paid particular attention to development of the odin DSL which makes models easy to develop and modify, a close interface with R which simplifies the flow of data in and out of models, and added efficient parallelisation which decreases runtimes while simulating stochastic models correctly.

Methods

Implementation

State space models relate input, output and state over time using a probabilistic model linking states at subsequent time steps, and can be used to model a broad range of processes. Below we describe how four major components of our framework, bundled as packages for the R programming language, can be used to implement these models. All of the packages require 100% code coverage of tests, and include unit tests designed by software engineers and integration testing (of entire analysis pipelines) designed by statisticians and epidemiologists.



Functions:
- Convert an odin model into a dust object
- Create a dust model on disk for use in a package

R commands:
```
- model <- odin_dust()
- odin_dust_package()
```

Functions:
- Create a new model instance: a dust object containing *n* particles
- Run a model forward and return the final state
- Run a model forward and return the state along the whole time series
- Change model parameters and move back to the first step
- Set the elements of the state to return from `run` and `simulate`

R commands:
```
- model$new()
- model$run()
- model$simulate()
- model$reset()
- model$set_index()
```

Functions:
- Read observation data in
- Run a particle filter (SMC) using a dust model
- Infer posterior parameter values with pMCMC
- Forecast trajectories from posterior parameter estimates
- Maximise likelihood with iterated filtering

R commands:
```
- data <- particle_filter_data()
- particle_filter$new(data, model, ...)
- pmcmc()
- pmcmc_predict()
- if2()
```

Figure 1. Overview of the odin.dust, dust and mcstate packages. For a typical user, a model will be written in the odin DSL, and the odin.dust package used to convert this into a dust model (which is an R object). This object can be used directly with functions provided by the dust package to run the model forward across a time series. With simulated or observational data, the mcstate package can be used to fit the model to this data using various techniques based around sequential Monte Carlo (particle filters).

Code to reproduce this analysis and all of the plots in this paper can be found at https://github.com/mrc-ide/odin-dust-plots.

odin – A DSL for R programmers to write efficient state space models

Packages which allow users to implement their own model code straddle a difficult dichotomy: making models fast and efficient to use, and making models fast and efficient to develop. Using an intermediary DSL is one approach to this issue, also applied in the popular JAGS and stan packages, as this offers a bridge between the more familiar language style used to develop models, and the compiled languages preferred for running large models[11]. Compared to writing directly in a compiled language, DSLs have the further advantage that we can design error messages which are domain specific, rather than the sometimes convoluted errors from compilers (which are necessary due to their generality).

odin is a DSL we have been developing since 2016, and has been used to model both continuous and discrete time models. odin is syntactically similar to R, and by taking advantage of R's 'non-standard evaluation'[12] presents users with a simple interface for describing sets of equations. The general approach of the package is that the user expresses their problems as a set of mathematical relationships, modelled as assignments to form a directed acyclic graph (DAG). odin then sorts that graph and transpiles the equations to code in a chosen target language.

Users can therefore write their equations in any order, which is more similar to mathematical formalism than declarative programming.

For use with ordinary differential equations (ODEs), odin transpiles to C code (https://mrc-ide.github.io/odin/) or JavaScript (https://mrc-ide.github.io/odin.js). For the models in this paper, we focus on transpilation of discrete time stochastic models into C++ using a framework (dust) that we describe below. In both cases, odin has an R interface, allowing its standalone use, or inclusion in R packages.

By eliminating logic and 'general programming' (such as defining types, writing loops), the models become relatively simple sets of mathematical truths that map closely to the scientific domain, yet remain efficient to solve. In addition to scalar relationships, odin provides a syntax designed to easily add structure to compartments - for example to represent age or transmission classes and the interaction between these without requiring explicitly written loops. Arrays representing structure classes are written implicitly with indices, meaning models can easily be extended. An example of adding age structure is shown in the Use Cases.

Specific functions available to the user beyond basic arithmetic include the random number generation functions detailed below, and optimised sums over state items. We include most of the functions available in the Rmath library[8] in odin, which are documented at https://mrc-ide.github.io/odin/articles/functions.html. A subset of these functions are currently available for dust, and we intend to continue to expand this support. If a required function is not available in odin, it is possible to write user-supplied C++ functions to be included in the model code. The only requirement for these functions is that they must return a scalar, and cannot modify the data they are passed. This is similar to the Csnippet approach taken by the pomp package.

mcstate – An R package implementing common SMC inference techniques, using odin models

The odin DSL gives modellers the ability to write a fast state space model in R, and interact with it in a number of fundamental ways. While some users may wish to implement their own inference techniques using these building blocks, we expect that most will use the standard methods we provide and test in the mcstate R package, as mcstate provides all the necessary routines for statistical inference from these models.

The key additional programmatic elements that mcstate provides for state space modelling are the definition of some observed data, and an observation function, which defines the log-likelihood of the observed data given the underlying model state. As the observation function is written directly in R, the user is free to define this however they choose, as long as it accepts the model state and some data as arguments. This may therefore use the state history stored in R, for example to compute the change in sizes of compartments and take advantage of large library of built-in functions. Some typical observation functions are described in the Use Cases below.

The mcstate package provides a particle filter implementation, also known as SMC (Sequential Monte Carlo)[13,14], which enables efficient parameter inference with high-variance model runs. A dust model is run forwards in time for a number of particles (n). At each step where observations are available these are compared to the data, and a likelihood weight computed for each particle. The n particles at observation time step j are then resampled with replacement, with probabilities corresponding to their likelihood weight, to select n particles to be run to observation time step j+1[15]. This SMC process runs the update function for all particles, resamples, and continues until the final observation is reached. This final state is sometimes referred to as a 'nowcast'. A function is provided to convert observational data into the correct format for the particle filter.

With this technique for combining potentially stochastic observations with a stochastic model, a marginal likelihood given model parameters can be produced. We can use the log-likelihood derived from the particle filter to perform Bayesian inference on model parameters. To do this, mcstate uses Markov Chain Monte Carlo (MCMC) over runs of the particles filter, known as pMCMC[16,17]. Currently mcstate supports standard Metropolis-Hastings MCMC: m_c independent chains are run, with care taken to ensure independent random number streams. The user provides prior distributions as functions in R, and a variance-covariance matrix for the proposals. Proposals at each step are drawn from a multivariate normal distribution, are reflected if outside of a specified minimum or maximum, and discretised if required. Alternatively, users can choose to apply iterated filtering, known as IF2[18]. This algorithm moves parameters in a random walk at each step in the time series, using the same weighting as the particle filter to select parameters for the next step. Over the course of the time series, and multiple iterations of this algorithm, this maximises the likelihood, and parameters approach values most compatible with the data.

We also support forecasting from the final observation position. The estimate of the posterior distribution produced by an MCMC run is sampled from by sampling particles with replacement. These runs extend the original model run by applying to state update functions until the time has reached the required length, picking up from the final random number generator (RNG) state of the particle. If further forecasts are required, a new RNG state is seeded from R's internal state, to avoid producing identical forecasts from the same streams of pseudorandom numbers.

dust – A C++ template library for driving parallel stochastic models from R

The above statistical techniques are computationally demanding, particularly as models become more complex, and to run them in a reasonable time-frame we needed an efficient engine to run stochastic models. Noting that between each observation step every particle is independent, we designed a system that could simulate the particles in parallel, so that we could take advantage of the increasing availability of multi-core CPUs. Our solution is implemented in the R package dust. The dust code itself is written in C++ with an R interface, which being a compiled language is typically faster to execute, and allows more careful memory management. To interface with dust, users must provide an update function, which is the core of the model, and we provide some examples below. While the user can write this in C++, most easily by modifying one of the included model examples, we expect most users to use odin.dust to generate this code automatically.

dust uses two main abstractions to represent and run state space models: Particle and Dust. A Particle object is a single trajectory simulated from the model and a Dust object is a collection of n_p Particles with the same model parameters and initial conditions, but different trajectories due to the inherent stochasticity of the system being simulated. Internally, Particles within a Dust object can be shuffled and sampled, to support particle filtering methods. A Dust object can be run forward in discrete steps, moving all Particles forward the same number of steps. This is the main computation in dust, and is parallelised on up to n_p CPU cores using OpenMP[19]. Using the static schedule to evenly distribute particles across cores gave close to linear increase in performance with the number of cores with long running models described in the Use Cases (Figure 2). We designed dust with these abstractions as this design has the advantage of having a high-level R interface directly designed to

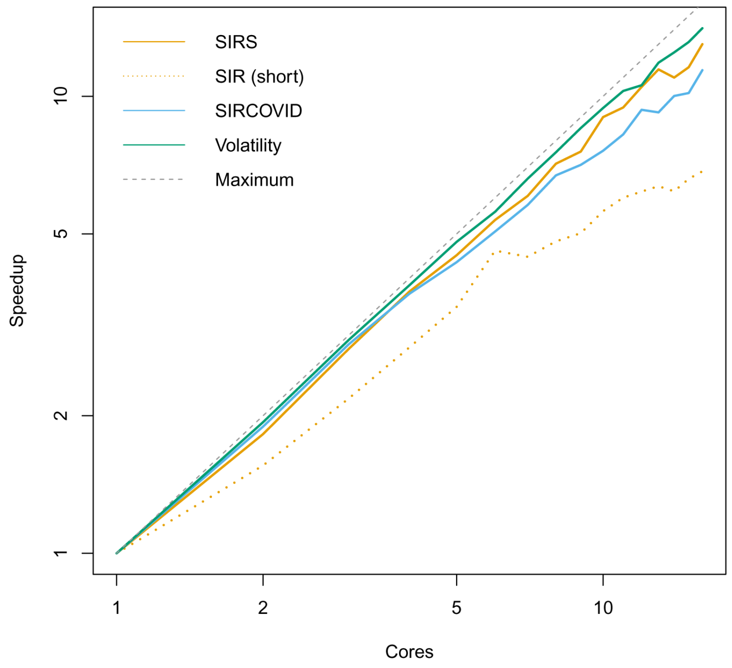


Figure 2. Speedup of dust simulations as number of threads increases, on a log-log scale. Models are described in the Use Cases. Speedup is defined as the ratio of wall time (total program time) taken to the wall time using a single thread. Here, a straight line along y = x would signify reaching the theoretical optimum of 100% parallelisation efficiency. The closer to the straight line, the closer to full efficiency at that number of cores. The models were run for 5×10^5 steps using $n_p = 10^3$ to artificially increase the amount of computation. The SIRS model has an additional R to S transition to make the infection endemic, otherwise the infection dies out, and no significant processing is used. The 'SIR (short)' model demonstrates a fall in performance for short running models, in this case with 10^3 steps and $n_p = 10^2$. We also ran the SIRCOVID model inference with $n_p = 10^2$ for 10^3 MCMC steps. The consistent speedup demonstrates that the multicore use is effective, even when running a full pipeline with a particle filter and evaluation of a log-likelihood in R.

work with particle filters, but still allowing control of individual particles in the lower-level C++ interface to develop new methods which operate on trajectories in different ways. This was useful when developing a 'simulate' method to run projections forward where every particle has a different set of parameters. For a technical audience, the "design" vignette describes why these decisions were made from a developer perspective.

In addition, we tested the speedup of a large SEIR (susceptible-exposed-infected-recovered) model for COVID-19 transmission in the UK, implemented using the `odin` DSL, using `dust` and `mcstate` to infer its parameters. Due to 19 age-classes which add structure to most of the model compartments, the model has around 1000 compartments in total. The computation time of running simulations of this model is therefore dominated by many random number draws, and so can be efficiently parallelised using the techniques described above. We confirmed this using a CPU profiler, finding that at least 61% of processing time being spent in the rbinom() function (described below). This dust model and its interface is named `sircovid`[20], and its code and documentation can be found at https://mrc-ide.github.io/sircovid. Running with $n_P = 100$ for 2000 steps, one MCMC chain on a laptop took around 1 hour with a single core, and showed roughly linear speedup with number of cores when used for either extra particles or extra chains.

On a personal computer, users can employ up to p cores to run the n_P particles of dust objects, or use these cores to run the m_c chains of mcstate, as long as $n_P m_c <= p$. On distributed infrastructure with disconnected nodes each with many cores, memory is shared on a node, but not between nodes. In this case, the optimal use of resources is typically to run n_P particles on a single node, using all p of its cores. The m_c chains, or parameter sets, can be independently run on up to m_c separate nodes, and their results combined using the provided functions. In some cases, parallelisation over particles is less efficient than over whole particle filters, for example when varying initial conditions and RNG seeds, on operating systems with poor OpenMP performance, or when evaluating model likelihoods at over a range of parameters sets. This can easily be automated in the pMCMC by using multiple 'workers' in mcstate, to parallelise m_c chains first, then use remaining threads to parallelise particles within these. Further flexibility is available to parallelise particle filters over multiple nodes, and collect results at the end, so users can pick a parallelisation mode that is most efficient for their problem. This is detailed in the 'parallelisation' vignette in the mcstate package.

The C++ source of the package exists largely as a header-only template library. This has the advantage that no platform-specific library code is needed for the generation of models, simplifying the installation process and giving wider support across operating systems and hardware architectures. Furthermore, compiling and optimising is always done using the whole model code in a single unit. The compilation is launched from within R, and once finished gives a shared object with R methods to run, shuffle and extract state from the underlying particles.

If writing a model directly in C++ there are some minimal interface requirements, which constrain the types of model which can be run through dust. Specifically, the user must provide a model class to dust with the following functions:

- `initial()` – Loads data passed from R to set the initial state and model parameters.
- `size()` – Computes the size of the model state for a single particle (number of compartments).
- `update()` – Updates the model for a single timestep. This may only depend on the previous state i.e. it must be Markovian. The function has access to the model parameters, timestep and random number generator functions.

More flexible simulation runs than provided by mcstate, for example from running counterfactuals, are straightforwardly supported by direct use of the dust object in R, while providing alternative parameter sets.

Random number generation

Generating random numbers from common distributions is a cornerstone of designing the model update function for many epidemiological models. This is also true computationally – when profiling a complex compartmental model for COVID-19 transmission, we found that at least 61% of program time was spent computing random deviates. R's default number generator is not able to operate in parallel, which meant that using the standard library functions for generating random deviates from common distributions was not an option. We therefore took particular care with the design of a parallel random number generator used by dust.

Random number generation on a computer produces a stream of pseudorandom numbers, usually integers in a specified interval, which are uncorrelated, but deterministic given a set starting point (the 'seed'). For stochastic model simulations there are two main considerations: running independent model realisations, and making results reproducible. Using the same seed for different particles will give identical results, and will break the assumptions of downstream inference methods[21]. However, using the same seed for an entire set of particles is desirable, so results can be reproduced. We also wish for our RNG implementation to 'play-fair', which means that results are independent of the specific hardware used, and the degree of parallelisation[22]. This is needed for scientific reproduciblity and effective debugging – knowing a change in results is attributable to a change in model, and not a change in the sequence of random number draws used to simulate the model is vital for model development.

The simplest solution, which is the default in R, is to run particles serially with each subsequent particle continuing from a single random number stream. However, serial particles places a limit on parallelisability. A frequent way around this is to make m RNGs for each parallel thread, and seed each one with a new but pre-specified seed. However, as the state space of the RNGs is highly non-linear and chaotic, it isn't possible to predict at which point a given seed will enter the stream, shared between all RNGs with the same design. For even modest simulation lengths and levels of parallelism, this can lead to correlated streams of random numbers, again breaking downstream inference assumptions.

One solution is to create p RNGs which can be advanced or 'jumped' a set number of steps. If each thread's RNG consumes k random numbers, then advancing each RNG p_i by ik steps before running will ensure independent streams for each process. If k is not known, this becomes more difficult. The approach we follow here uses a new class of RNGs known as Xoshiro (XOR, shift, rotate)[23]. These generate a stream of pseudorandom integers in the interval $[0,2^{64})$ with a period 2^{256}. Generation is very fast, but importantly also implements a `jump()` function which advances the generator by 2^{128} steps in a time comparable to a single random number draw. Applying this allows the initialisation of up to 2^{128} RNGs, each capable of drawing a stream of up to 2^{128} pseudorandom numbers before correlating. These concepts are summarised in Figure 3.

Random number streams should not vary based on the number of threads used in a specific run, and should instead always be reproducible from the same single seed. Therefore every particle p has its own RNG, rather than every thread; this is feasible given the relatively small state (256 bits) used by Xoshiro compared with generators such as Mersenne Twister (2560 bits)[24]. A single 64-bit integer seed is passed from R, with the remaining three chunks of 64-bits-of-state set pseudo-randomly from this using the splitmix64 algorithm[25]. These 256 bits are used to set the initial state of a `xoshiro256**` generator. The RNG state for particle i begins with this state, after applying the `jump()` function i times.

Many off-the-shelf parallel random number generators are aimed at repeated generation from a distribution with the same parameters, including those in the C++ standard library. This is ill-suited to state space models, where distribution parameters typically change between every generation, and between every particle. The

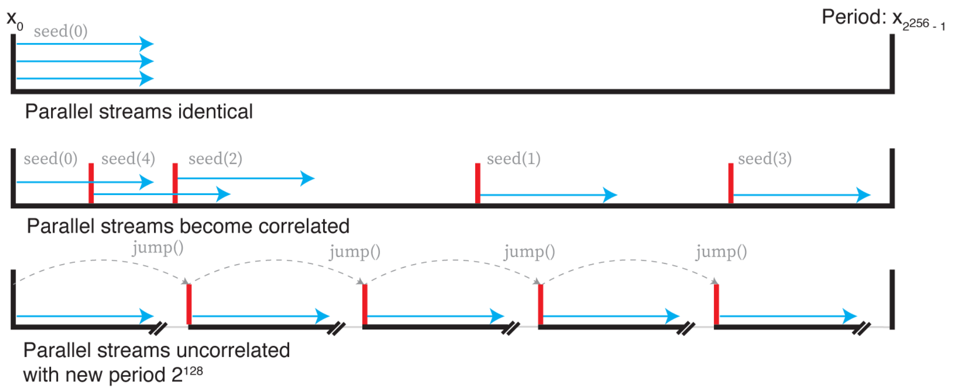

x_0
seed(0)
Period: $x_{2^{256}-1}$
Parallel streams identical
seed(0)
seed(4)
seed(2)
seed(1)
seed(3)
Parallel streams become correlated
jump()
jump()
jump()
jump()
Parallel streams uncorrelated
with new period 2^{128}

Figure 3. Top: parallel random number streams with the same seed are identical. Middle: parallel random number streams with different streams can quickly become correlated. Bottom: using the `jump()` function of the Xoshiro generator moves forward 2^{128} steps, giving evenly spaced, uncorrelated streams of random numbers.

TensorFlow[26] code base contains suitable implementations, but would be a large and complex dependency, difficult to make fully compatible with R. Therefore, we added code which uses the Xoshiro generator to transform into random deviates from statistical distributions:

- runif(a, b) - A uniformly distributed real number in the interval [a, b), by dividing RNG state by its maximum value of 2^{64}.
- rnorm(μ, σ) - A normally distributed real number with mean μ and variance σ^2, by applying the Box-Muller transform to runif(0, 1)[27].
- rbinom(n, p) - A binomially distributed integer given n trials and a probability of success p. Uses inversion transform sampling with exponentiation by squaring if $np < 10$[28], or transformed rejection sampling with the 'BTRS' algorithm otherwise[29].
- rpois(λ) - A Poisson distributed integer given rate λ. Uses Knuth's algorithm for $\lambda < 10$[30], and transformed rejection sampling otherwise[31].

All of these methods are optimised for when the parameters of the distribution change every sample, as expected with stochastic state space models. We plan to add other random number distributions as required, though these were sufficient for all models currently tested. This is all implemented using C++ to be bundled with dust, and can directly be accessed from R.

Operation

Given a model which is known to be a good description for the data under study, we provide an overview of the typical workflow runs through these packages in sequence. Steps 1 and 2 are model simulation, 3 to 7 for model inference, and 8 for forecasting. In cases where models are not being fitted to data, a dust generator can be used directly to simulate from the model with given parameters, and this process ends after the second step. If any model parameters are being inferred from the data steps 1–7 are required, with step 8 an optional addition if a further forecast using the inferred values is needed. Users with more complex needs not met by an odin model may also skip step 1, and write a dust target in C++ directly, while still using the dust RNG library and functions if required.

1. *Write a model in the odin DSL*. Markov models will define a set of `update()` functions which together give the state at t +1 by operating on the state at t.
2. *Compile the model into a dust object*. Using odin.dust, this will create a shared library and R interface. This will turn the odin code into a single `update()` update function usable by dust. This step may be performed iteratively within an interactive session for rapid development, or by creating an R package for more robust development.
3. *Write an observation function*. This will compare model simulations to data, and calculate the log-likelihood of the model run given the data. Users are free to implement this however they wish, and can leverage any R function or package to do so.
4. *Load observed data*. Typically this will be time series data in R, as a `data.frame.` Use the included functions in mcstate to convert this into input for the particle filter, potentially adding an offset and intermediate steps without observations.
5. *Create a particle filter object*. This uses the dust generator, observation function and observed data.
6. *Set parameters*. Define prior functions and pMCMC jump size for unknown parameters; set the values of known parameters.
7. *Run a pMCMC*. Using parameters and the particle filter, this will return posterior density estimates for each unknown parameter.
8. *Forecast trajectories*. If a forecast is required, run the `predict()` function in mcstate after the pMCMC, which will create simulated trajectories for each particles past the end of the data, while sampling parameters from the posterior.

In some cases, users will want to first undertake model criticism, where possible models are compared in their ability to describe the data being studied, and iteratively improved. This may also be used as a replacement where the computational demands of a full pMCMC run are too great. In this case, steps 1–5 are the same, but then a maximum likelihood method replaces subsequent steps, and priors are only optionally specified:

- 6a. *Investigate model specification*. Use maximum-likelihood estimation with IF2 to determine features of the data which are incompatible with the model.
- 7a. *Compare possible models*. Likelihood estimates from IF2 can be produced by running sets of particle filters with the `sample()` function. For nested model structures, these can be compared with likelihood-ratio tests.
- 8a. *Use this information to design a better model*. Changes to the model structure can be made, returning to step 1 iteratively until an appropriate model is arrived at.

System requirements for running are:

- R (>=v3.5.0)
- A C++ compiler such as gcc or clang is needed to compile dust models.
- An appropriate OpenMP library (>=v3.0), including a C++ compiler supporting the `-fopenmp` option, such as gcc (>=v4.0) or clang (>=v3.9). If OpenMP is not available, models will still compile, but parallelisation of particles will not be supported.
- odin, odin.dust, dust and mcstate, all of which can be installed with standard tools. Here, we used odin v1.1.12; odin.dust v0.2.7; dust v0.9.3; mcstate v0.6.0.

All our R packages are available for R on Linux, OS X and Windows, and are open source using the MIT licence.

Use cases

Here we exhibit some brief examples of state space models, followed by a typical use case in epidemiology. We demonstrate the ease of use of these packages as this model becomes more complex, and is expanded to more realistic scenarios. These examples are included with the dust package, and detailed vignettes to reproduce the analysis here are included with the mcstate package.

Basic stochastic models

A basic model for volatility, which is a broadly used concept in finance describing randomly distributed variance of an asset's price *x*, is given by:

$$dx_t = axdt + \sigma dW_t$$

where α is the constant drift, σ is a constant volatility and dW_t is a Weiner process with zero mean and a variance of one[32,33]. Given the properties of Weiner processes (Brownian motion), the update function using the Euler-Maruyama method is:

$$x_{t+1} = ax_t + \sigma * w$$
$$w \sim N(0,1)$$

which in the odin DSL is simply:

```
update(x) <- x * alpha + sigma * rnorm(0, 1)
initial(x) <- x0
alpha <- user(0.91)
sigma <- user(1)
x0 <- user(0)
```

The `user()` syntax specifies this will be a parameter provided through the R interface, either directly or through an inference method (such as mcstate). Default values can be set in brackets. Multiple realisations of this model can be run through dust (Figure 4) as follows:

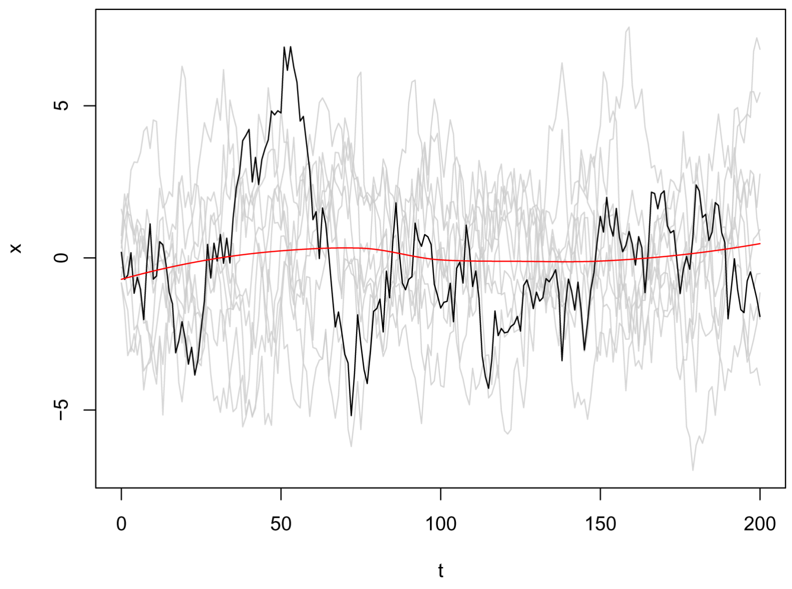


Figure 4. Plot of the *x* state from ten independent realisations (particles) from the volatility model, with an example trajectory highlighted in black, and the smoothed mean value from all particles in red.

```
gen <- dust::dust_example("volatility")
# more generally:
# gen <- odin.dust::odin_dust("volatility.txt")
```

```
vol <- gen$new(data = list(alpha = 0.91),
               step = 0,
               n_particles = 10,
               n_threads = 4L,
               seed = 1L)
vol$run(10)
```

This is included as an example in the dust package, but users can load their own model code by calling `odin_dust` on the text file containing their model. This example illustrates running ten particles, ten timesteps forwards parallelised over four threads. The seed provided gives reproducible, uncorrelated runs for each of the particles.

Stochastic SIR model

Models of infectious disease transmission – such as the susceptible-infected-recovered (SIR) model[34,35] – are an obvious use of our packages. Typically these models can be expressed in three related forms: ODEs, stochastic differential equations (SDEs) or as a continuous time Markov chain (CTMC)[36]. Their solutions have different properties: ODEs are fast to solve numerically, and are deterministic given a set of initial conditions; SDEs give stochastic solutions each time they are solved which better represent the variance in real-world systems, and still have efficient numerical solvers; CTMCs best represent the discrete nature in small populations and correctly model absorbing states, but are computationally more intensive to realise trajectories from[37]. For the remainder of this section we focus on stochastic models, particularly CTMC formulations of infectious disease transmission models. A simple definition (using the ODE formalism) of the SIR model is:

$$\frac{dS}{dt} = -\beta\frac{SI}{N}$$
$$\frac{dI}{dt} = \beta\frac{SI}{N} - \gamma I$$
$$\frac{dR}{dt} = \gamma I$$

S is the number of susceptibles, I is the number of infected and R is the number recovered; the total population size $N = S + I + R$ is constant. β is the infection rate, γ is the recovery rate.

This model can be discretised in time steps of width $d\,t$ using the following update equations for each time step[36–38]:

$$\begin{aligned} S_{t+1} &= S_t - n_{SI,t} \\ I_{t+1} &= I_t + n_{SI,t} - n_{IR,t} \\ R_{t+1} &= R_t + n_{IR,t} \end{aligned}$$

where the number of individuals moving between compartments are given by drawing from binomial distributions:

$$n_{SI,t} \sim B(S_t, 1 - \exp(-\beta \frac{I_t}{N} \cdot dt))$$

$$n_{IR,t} \sim B(I_t, 1 - \exp(-\gamma \cdot dt))$$

the binomial distribution is used as there are n trials, one for each individual in the compartments, who move with per-capita transition probability p. In a single time step, p can be calculated as $1 - e^{\lambda \cdot dt}$ where λ is the transition rate, as in a Poisson process time between events is exponentially distributed.

These equations can be written in the odin DSL as:

```
## Definition of the time-step and output as "time"
dt <- user(1)
initial(time) <- 0
update(time) <- (step + 1) * dt
```

```
## Model parameters (default in parenthesis)
beta <- user(0.2)
gamma <- user(0.1)
```

```
## Initial conditions
initial(S) <- 1000
initial(I) <- 1
initial(R) <- 0
```

```
## Core equations for transitions between compartments:
update(S) <- S - n_SI
update(I) <- I + n_SI - n_IR
update(R) <- R + n_IR
```

```
## Individual probabilities of transition:
N <- S + I + R # total population size
p_SI <- 1 - exp(-beta * I / N * dt) # S to I
p_IR <- 1 - exp(-gamma * dt) # I to R
```

```
## Draws from binomial distributions for numbers changing between
## compartments:
n_IR <- rbinom(I, p_IR)
n_SI <- rbinom(S, p_SI)
```

This would be saved as `sir.R` and then compiled with `odin.dust::odin_dust ("sir.R").`

Initial conditions here are fixed, but they can also be added to the `user()` group to be set from R for each run. For infectious disease models, users may follow the guidance in the odin documentation on discretising ODE models using appropriate random number draws (https://mrc-ide.github.io/odin/articles/discrete.html).

This model can simulate an epidemic forward in time using fixed parameters, as shown in Figure 5. We can also use this model to demonstrate the inference process (steps 3–8 in Operation), showing how the transmission rate β and recovery rate γ (as well as $R_0 = \frac{\beta}{\gamma}$) can be inferred from observed daily case counts y_t. For each day t, the number of expected new cases (if all cases are observed) can be assumed to be Poisson distributed with mean y_t, so an observation function can be written by taking the log of a Poisson probability mass function at $k_t = S_{t-1} - S_t = n_{SI,t}$. This can easily be achieved using the builtin R function `dpois()` (one of many probability-distribution functions available to users):

```
case_compare <- function(state, prev_state, observed, pars = NULL) {
  cases_modelled <- prev_state[1, , ] - state[1, , ]
  dpois(incidence_observed, observed$cases, log = TRUE)
}
```

In the state array, the first dimension is over model compartments, the second dimension is over particles, and the third over time. So the index '1' extracts the first state, the number of susceptibles. Multiple data-streams can straightforwardly be added by noting that log-likelihoods sum, as long as they are conditionally independent. That is, given the simulated quantities in the model, the observed quantities are assumed to be independent, and have independent noise. So a similar log-likelihood component could be defined based on deaths, in a model with these compartments, and added to the existing function. If a data stream isn't measured at a particular time point, it simply contributes zero to the log-likelihood.

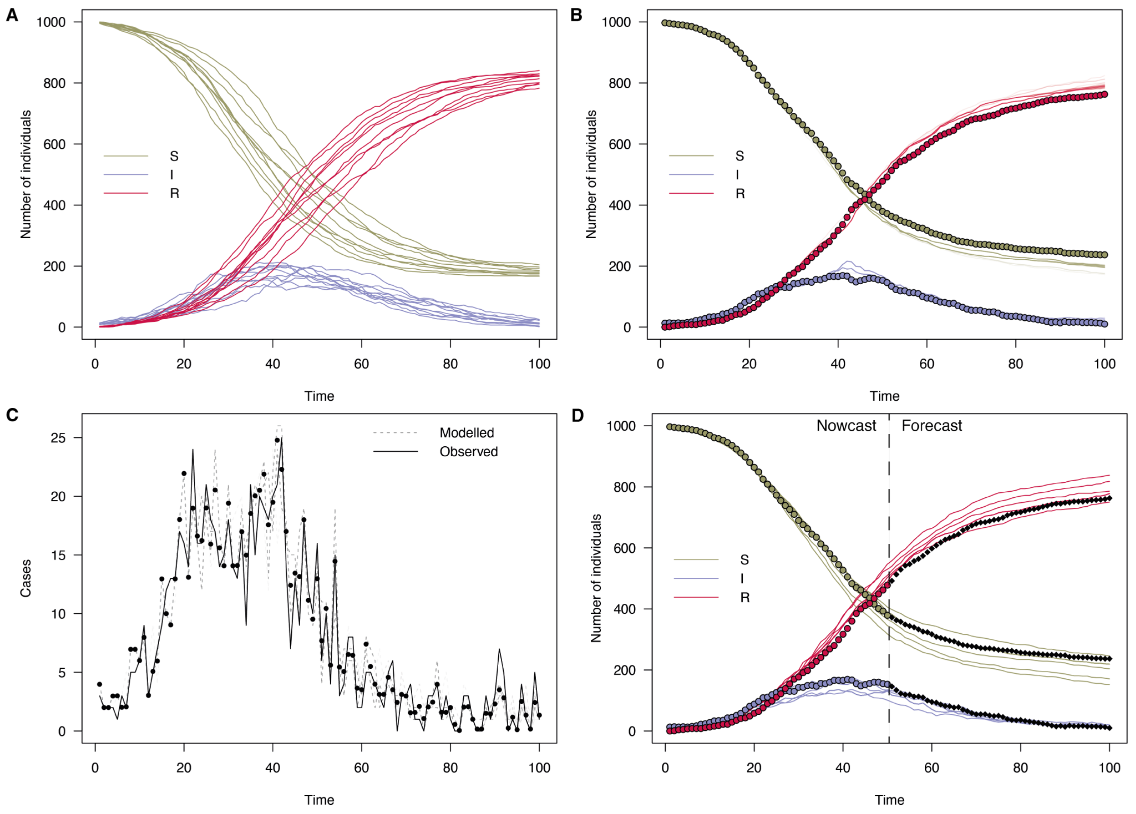


Figure 5. Plots of the number of individuals in the *S*, *I* and *R* compartments over time in an SIR model with β = 0.2 and γ = 0.1. A: 10 particles run forward for 100 time steps. **B**: Solid points are data generated from the model from which simulated case counts were produced. Lines are 100 particles run forward using a particle filter with this data. **C**: Daily case incidence data which was fitted to (black), and modelled incidence of 100 particles (grey). Average of particles shown as points **D**: Extending the particle trajectories forward in time by simulating the model forward with parameters sampled from the estimated posterior. Here, only the first half of the time series was used to show a more uncertain part of the epidemic, subsequent real points are shown in black.

A particle filter can then be set up using mcstate (steps 4 and 5), choosing the number of particles, and formatting observational data appropriately with the built-in function `particle_filter_data()`. The observations must be evenly spaced, though missing observations are permitted. The step size in the data is defined to be one, and $\frac{1}{dt}$ update steps are taken between each observation.

```
n_particles <- 100
dt <- 0.25
data <- particle_filter_data(data, time = "day", rate = 1/dt)
filter <- particle_filter$new(data = data,
                              model = sir,
                              n_particles = n_particles,
                              compare = case_compare)
```

The right panel of Figure 5 demonstrates using this observation function with case data simulated from the model. Only those trajectories consistent with the data are continued forward at each step, and a final log-likelihood of the model parameters given the data is produced.

Steps 6 and 7, which are used to estimate the posterior density for β and γ, are achieved by defining sampling distributions for the parameters, and running a set of MCMC chains. Priors for each parameter can be added at this stage, with an example shown on γ for demonstrative purposes:

```
beta <- pmcmc_parameter("beta", 0.2, min = 0)
gamma <- pmcmc_parameter("gamma", 0.1, min = 0,
                         prior = function(p)
                           dgamma(p, shape = 1, scale = 0.2, log = TRUE)
                        )
proposal_matrix <- diag(2) * 0.01^2
mcmc_pars <- pmcmc_parameters$new(list(beta = beta, gamma = gamma),
                                  proposal_matrix)
```

```
pmcmc_run <-
  pmcmc(
    mcmc_pars,
    filter,
    n_steps = 2000,
    save_state = TRUE,
    save_trajectories = TRUE,
    progress = TRUE,
    n_chains = 4
  )
```

Noting that in the SIR model $R_0 = \frac{\beta}{\gamma}$, we could alternatively directly sample R_0 instead of β. This is achieved by also supplying a transformation function, which takes parameters being sampled in the MCMC, and returns a list of user input parameters needed by the odin model:

```
R0 <- pmcmc_parameter("R0", 2, min = 0)
parameter_transform <- function(pars) {
    beta <- pars[["gamma"]] * pars[["R0"]]
    gamma <- pars[["gamma"]]
    list(beta = beta, gamma = gamma)
}
```

Parameter transforms can also be used to represent time-varying parameters. One way of doing so is to make fixed time points as pmcmc_parameters in R, define a piecewise linear interpolation between fixed parameters in the transformation, and give a dust model a dense vector with the parameter value at every time point. We refer interested readers to the example of the time-varying β parameter in the SIRCOVID package.

Posterior distributions for β and γ are available from the `pmcmc` object, and are plotted for this example in Figure 6. They can be loaded directly into standard MCMC analysis packages such as `coda`[39] to produce diagnostics such as effective sample size, the Gelman-Rubin diagnostic $\hat{R}$. As is typical with MH sampling, the proposal distribution will likely need to be tuned to get an appropriate acceptance rate, typically thought to be 0.234 for high dimensional problems[40]. One algorithmic way to do this is to run the chains for a short time, calculate the variance-covariance matrix among the samples, and use this as the proposal kernel. This process is covered in the 'Tuning the pMCMC' section of the SIR models vignette in the mcstate package.

Finally, producing a forecast past the end of the data (step 8) is achieved simply by calling `predict()` on the above MCMC object (as shown in the final panel of Figure 5):

```
forecast <- predict(pmcmc_run,
                    steps = seq(400, 800, 4),
                    prepend_trajectories = TRUE,
                    seed = pmcmc_run$predict$seed)
```

Maximum-likelihood estimation with the SIR model

Using the particle filter constructed above, an iterated filtering algorithm can be applied to find parameter values which maximise the model likelihood. This is useful for the model criticism workflow outlined in steps 6a-8a. This may also be useful when pMCMC would stretch the available computational resources, as evaluating the likelihood profile at different points has more favourable scaling properties[41–43]. In this mode, the user sets parameters up as for pMCMC, and adds a control object including an independent perturbation strength for each parameter `pars_sd`, a population size of parameters `n_par_sets`, and a cooling schedule in `iterations` and `cooling_target`:

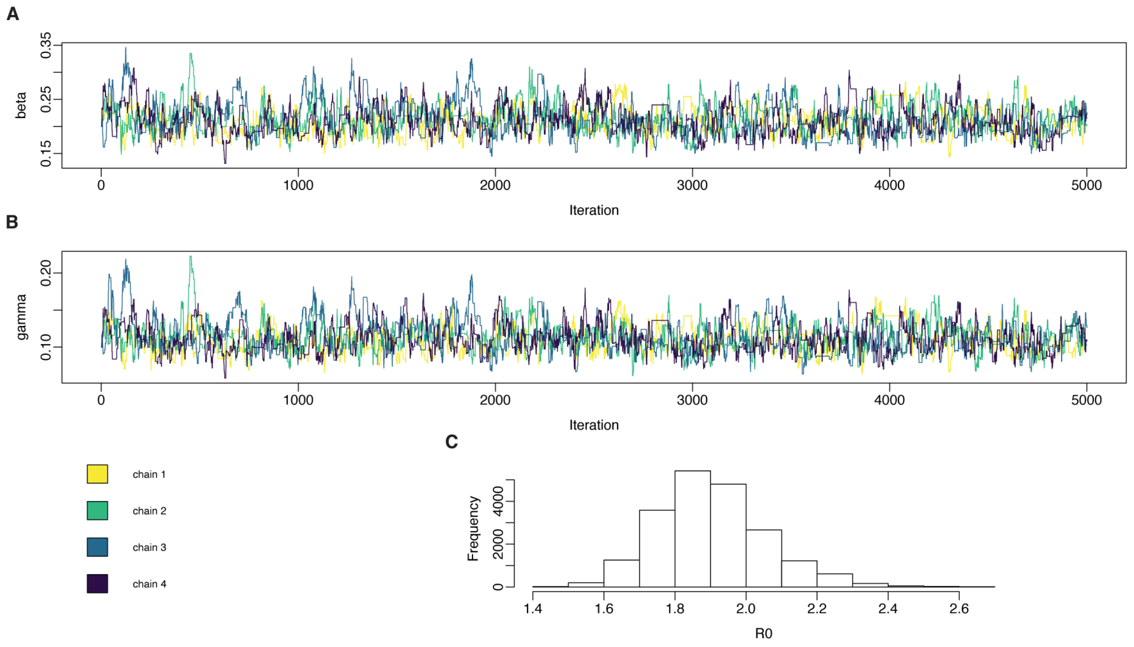


Figure 6. Inferring parameters in compartmental models: the SIR model fitted to simulated daily case data. The particle filter was set up as specified, and four independent chains were run, each chain taking 2×10^3 samples for the SIR model. In the SIR model, true values are $\beta = 0.2$, $\gamma = 0.1$, $R_0 = 2$. **A**: posterior samples from β. **B**: posterior samples from γ. **C**: marginal posterior distribution for $R_0 = \frac{\beta}{\gamma}$.

```
pars <- mcstate::if2_parameters$new(
            list(mcstate::if2_parameter("beta", 0.5, min = 0, max = 1),
                 mcstate::if2_parameter("gamma", 0.01, min = 0, max = 1)))

control <- mcstate::if2_control(
  pars_sd = list("beta" = 0.02, "gamma" = 0.02),
  iterations = 100,
  n_par_sets = 300,
  cooling_target = 0.5)

res <- mcstate::if2(pars, filter, control)
```

This will yield a likelihood which is maximised over the course of the iterations, and the corresponding parameter estimates. The likelihood and its error can be estimated by running particle filters at these parameter estimates using the `if2_sample()` command. This mode is covered in more detail in the "if2" package vignette.

Prior specification and prior predictive checks

Above, we showed a Gamma(1, 0.2) prior on γ for demonstrative purposes. The default if no prior function is specified, is to use a flat improper prior, which does not contribute to the posterior. For model features which do not fit the data, it may not be possible to set a suitable prior, so this may be a particularly useful when undertaking model criticism (steps 6a–8a). The prior function is completely flexible, and therefore allows any functionality the R language allows. This can therefore include more complex setups such as using tools from external packages, or drawing from another model's posterior, which may be used to implement hierarchical models.

When setting priors, users may wish to perform prior predictive checks to determine whether the chosen prior functions are appropriate for the model and data[44]. First, parameters are selected by drawing from the priors (in the example below, from a uniform distribution for β. and gamma distribution for γ). Then, the model is simulated forwards using these parameters, and the resulting data series plotted to determine appropriateness. The simulate method in dust can be used to run across a whole time series and return the model state:

```
step_start <- 0
step_end <- 100
mod <- sir$new(pars = list(), step = start, n_particles = 100L)

# Note here the "r" version of the "d" distribution densities used in the
# prior functions. Care should be taken translating more complex densities
# into random draws
prior_draw <- list("beta" = runif(1, 0, 1), "gamma" = rgamma(1, 1, 0.2))
mod$reset(pars = prior_draw, step = start)
prior_data <- mod$simulate(step_end)
```

Adding age-structure to the SIR model

This model can be extended to add more flexibility or more specificity when modelling the biology of different diseases. For example: adding new compartments to represent other disease states; compartments which represent spatial or age structuring of the population; or delay distributions which model different rates at which individuals pass through disease pathways. By matching the compartments and the transitions between them to the disease being studied, infectious disease epidemiologists can flexibly and accurately model a wide variety of real world processes.

As an example of how the basic code given above can be extended, we demonstrate how the SIR model can incorporate age-structure into each of its three compartments. Adding age structure to the model consists of the following steps, which turn variables into arrays:

- Define the number of age categories as a user parameter N_{age}.
- Add age structure to each compartment, by adding square brackets to the left hand side of each declaration in the odin definition of the model given at the beginning of the section describing stochastic SIR model.

- Modify the right hand side of each declaration to use quantities from the appropriate compartment, by adding indices *i* and *j* as needed. These will automatically be turned into loops by odin.dust.
- Where an age compartment needs to be reduced into a single compartment/variable, we use `sum` (though further array reduction functions are available).
- Define the dimensions of all arrays, for example by setting `dim(S) <- N_age`.

Alone, this would simply give N_{age} independent processes equivalent to the first model, scaled by the size of the population in each age category. To actually make this useful, some form of interaction or transitions need to be added between the compartments. An example of this would be to add an age-specific contact matrix *m*, which defines a different force of infection λ for each age group. This is calculated by

$$\lambda_i = \frac{\beta}{N} \cdot \sum_{j=1}^{N_{\text{age}}} I_j m_{ij} \tag{1}$$

In the odin DSL:

```
m[, ] <- user() # age-structured contact matrix
s_ij[, ] <- m[i, j] * I[i]
lambda[] <- beta / N * sum(s_ij[i, ])
```

The contact matrix *m* is input from R. Its entries m_{ij} define the intensity of contact between age classes *i* and *j*. One choice is to base it on the POLYMOD survey, which can conveniently be loaded in through the `socialmixr` R package[45]. This can be provided as input to the model by adding it to the list returned by a transformation function.

The probability of infection of a susceptible is then indexed by this force of infection:

```
p_SI[] <- 1 - exp(-lambda[i] * dt)
```

Putting this all together, the key components of the age structured SIR model is as follows (omitting initial conditions and parameter values from the unstructured SIR model for simplicity):

```
## Core equations for transitions between compartments:
update(S[]) <- S[i] - n_SI[i]
update(I[]) <- I[i] + n_SI[i] - n_IR[i]
update(R[]) <- R[i] + n_IR[i]
```

```
## Individual probabilities of transition:
p_SI[] <- 1 - exp(-lambda[i] * dt) # S to I
p_IR <- 1 - exp(-gamma * dt) # I to R
```

```
## Force of infection
m[, ] <- user() # age-structured contact matrix
s_ij[, ] <- m[i, j] * I[i]
lambda[] <- beta / N * sum(s_ij[i, ])
```

```
## Draws from binomial distributions for numbers changing between
## compartments:
n_SI[] <- rbinom(S[i], p_SI[i])
n_IR[] <- rbinom(I[i], p_IR)
```

```
## Total population size
N <- sum(S) + sum(I) + sum(R)
```

```
# Array dimensions
dim(S) <- N_age
dim(I) <- N_age
dim(R) <- N_age
dim(n_SI) <- N_age
dim(n_IR) <- N_age
dim(lambda) <- N_age
dim(m) <- c(N_age, N_age)
dim(s_ij) <- c(N_age, N_age)
```

While we use 1- and 2- dimensional structures here, odin currently supports up to 8 dimensions, allowing for concise description of structured models (at the time of writing, we know of use of up to 4 dimensions).

Comparison with alternative packages

Although state space models can in some cases be analysed using a general Bayesian hierarchical framework such as JAGS[46] or stan[47], care needs to be taken with state space models as trajectories can rapidly diverge from time-series data with stochastic update functions, and fitting may become slow[48]. Two previous packages which aim to solve these issues in a similar way are pomp (partially-observed Markov processes)[9] and libBi (library for Bayesian inference)[10].

Both packages use similar concepts to dust and mcstate, requiring users to define a Markovian update function (`rprocess()` in pomp; `transition()` in libBi), an initial state (`rinit()` in pomp; `initial()` in libBi) and a observation function (`dmeasure()` in pomp; `observation()` in libBi). These packages both support simulation and inference from the model using the same overall methods as mcstate, and additionally support some optimisation procedures such as maximum likelihood or particle perturbation.

These packages also support some link between interpreted and compiled languages. In pomp, the interface is in R, but it also any of these functions may be written either in R or in C. This allows us to demonstrate the advantage of using compiled functions to simulate from models on an even footing. Using an SIR model coded in pomp using R functions, and and equivalent implementation in pomp coded using C function, a speedup of around 100x was seen comparing C to pure R, even for this simple model (Table 1). libBi uses a DSL to define both the update and observation functions, which are transpiled into C++, then compiled into a standalone executable

Table 1. Comparison of packages for SIR model implementations. This table lists some features of each package. The CPU time is for a run of 10^6 steps with $n_p = 10$; $S_0 = 10^6$; $I_0 = 10$; $dt = 10^{-4}$ on a single core. We focus on a long simulation time, as profiling of the more realistic COVID-19 transmission model showed most computation time during inference was spent in this stage, and minimises measurement of overheads. Array indexing can be automated when a DSL infers the correct 'C-Tran' sums and loops to add, or manual when they must be written in the model by the user. Bayesian inference algorithms include particle MCMC (pMCMC), approximate Bayesian computation (ABC)[49], SMC2[50] and sequential importance sampling (SIS)[51]; maximum likelihood inference algorithms include iterated filtering (IF2) and numerical likelihood optimisation (optim).

Package	Potential parallelisation	CPU time	Model language	Interface language	Data input	Array indexing	Parameter constraints/ transforms	Inference methods
dust	$n_p m_c$	0.96s	R/odin DSL	R	R	Automatic	Arbitrary R code	pMCMC, IF2
pomp	m_c	82s	R	R	R	Manual	Fixed choice of functions	pMCMC, IF2, ABC
pomp	m_c	1.3s	C	R				
libBi	$n_p m_c$	1.04s	libBi DSL	bash	NetCDF/ RBi	Automatic	In DSL	pMCMC, SMC^2, SIS, optim

supporting all its inference methods. We implemented an identical stochastic SIR model in all three packages. With compiled functions, all three methods run at similar speeds (Table 1).

The main differences between pomp and our packages are:

- In pomp, parallelisation is only over independent chains m_c; parallelisation of particles is unsupported.
- To write efficient code in pomp, users must write directly in C, no DSL is available. This also makes function debugging more challenging as there is no built-in parser.
- Automated generation of code for arrays is not supported in pomp. For multi-compartment models this requires the user to write loop indexes over arrays/tensors manually, sometimes known as 'C-Tran' code.
- Parameter constraints are not directly supported in pomp and must instead be implemented through monotonic transforms, such as taking the logarithm.

The main differences between libBi and our packages are:

- The interface to libBi is through the command line, and parameters are set through configuration files.
- Parameter setup to be used for both simulation and inference in libBi may require different model definitions.
- Combining inference and forecasting tasks is possible, but requires chaining configuration files in libBi rather than relying on object reuse.
- In libBi, Input and output is in the NetCDF format, which is efficient and compact, but requires extra tools and knowledge to manipulate, and is not human-readable.
- To interface with R, the RBi package can be used, which converts between NetCDF format and R objects, and constructs calls to the libBi command line interface.
- All functions must be written in the libBi DSL, and although this is extensive, it is still more restrictive than a full language such as R, as may be used in pomp and our packages. However, this also allows faster compiled code to be generated for the observation functions.
- An alternative method of parallel random number generation is used in libBi[52,53], and further parallelisation over parameters is additionally supported when using the SMC^2 algorithm for inference.

Overall we would summarise these three packages as being broadly similar in purpose and efficiency. We believe that the major advantages of our packages are the tight interface with R, the easy-to-use DSL, and the considered parallel simulation machinery. In comparing these packages, anecdotally we found the flow of data between the source and models was simplest in our packages – for our rapidly evolving COVID-19 model, this made our packages the only viable choice. The growing set of auxiliary functions specific to COVID-19 modelling in the SIRCOVID package are easily referenced between packages, demonstrating that being close to the language users are conversant with has a clear interface advantage. However, we expect that different users will have different preferences for these packages. Users will now have three good software choices available, which they can decide between based on their background and needs.

Summary

State space models are broadly used to model biological processes, particularly transmission of infectious diseases. In principle a simple state space model can readily be implemented in any number of programming languages, but keeping this model efficient, reproducible and correct, especially as it is expanded to include more processes and complexity, is a cross-disciplinary challenge. We have produced a suite of packages intended to make the mechanics of model development and fitting as simple and efficient as possible, so modellers can focus on the biology of their problem, rather than spending time on software integration challenges.

Software solutions must balance the competing needs of modellers, statisticians and software developers, thus tradeoffs necessarily exist. Anecdotal experience with expert modellers led to the following design requests: use of a DSL close to R to make compiled code; reproducibility of results; parallelisation with a fair random number draws for simulation; tight integration with the R language to allow easier definition of an observation function; fast implementation of new inference methods, especially when adding compartments or age-structure; and more flexible uses of the model simulator.

We aimed to produce methods which lie closer to the typical skill set and scientific interest of epidemiologists than previous state space modelling packages. Resulting model objects are lightweight and directly connected to R, making their reuse easy and flexible. No advanced programming skills are required to use the packages, and the definition of likelihood functions in R itself means that in practice few restrictions are placed on models, other than that they are Markovian. Optimised code for model simulation is automatically generated, and modern Bayesian methods such as SMC can be applied without needing a thorough understanding of the mechanics of their operation. Models are guaranteed to be reproducible, 'play fair' with randomness even when parallelised, and come with a suite of fully unit-tested inference methods. CPU parallelisation is efficient, and the code developed here will form the basis of future speedups using specialist hardware such as general purpose graphics processing units.

These packages helped us with reliability, speed of model development, and the speed of real-time inference in our model of COVID-19 transmission in the UK. Due to their generality, we believe these packages will be more broadly useful for a range of modelling attempts, and will mean modellers do not have to reinvent the wheel each time a new model and inference method is produced.

Software availability

Software available from:

- odin: https://mrc-ide.github.io/odin
- odin.dust: https://mrc-ide.github.io/odin.dust
- dust: https://mrc-ide.github.io/dust
- mcstate: https://mrc-ide.github.io/mcstate

Source code available from:

- odin: https://github.com/mrc-ide/odin
- odin.dust: https://github.com/mrc-ide/odin.dust
- dust: https://github.com/mrc-ide/dust
- mcstate: https://github.com/mrc-ide/mcstate
- Plots: https://github.com/mrc-ide/odin-dust-plots

Archived source code as at time of publication:

- odin: http://doi.org/10.5281/zenodo.4772403[54]
- odin.dust: http://doi.org/10.5281/zenodo.4772398[55]
- dust: https://doi.org/10.5281/zenodo.4772395[56]
- mcstate: http://doi.org/10.5281/zenodo.4772455[57]
- Plots: https://doi.org/10.5281/zenodo.4293396[58]

Software license: MIT

Acknowledgements

We are grateful to both the Imperial COVID-19 response team and the RESIDE group, who advised on the COVID-19 model and R integration respectively. We would also like the thank the Tensorflow developers, as we based our code for rejection sampling for binomial and Poisson draw on their implementation. The COVID-19 death data fitted to was provided by the Office for National Statistics, which is provided freely and openly under the Open Government Licence.

References

1. Grassly NC, Fraser C: **Mathematical models of infectious disease transmission.** *Nat Rev Microbiol.* 2008; **6**(6): 477–487.
PubMed Abstract | Publisher Full Text | Free Full Text

2. Whitty CJM: **The contribution of biological, mathematical, clinical, engineering and social sciences to combatting the west african ebola epidemic.** *Philos Trans R Soc Lond B Biol Sci.* 2017; **372**(1721): 20160293.
PubMed Abstract | Publisher Full Text | Free Full Text

3. Heslop DJ, Chughtai AA, Bui CM, *et al.*: **Publicly available software tools for decision-makers during an emergent epidemic-systematic evaluation of utility and usability.** *Epidemics.* 2017; **21**: 1–12.
PubMed Abstract | Publisher Full Text

4. Thompson RN: **Epidemiological models are important tools for guiding COVID-19 interventions.** *BMC Med.* 2020; **18**(1): 152.
PubMed Abstract | Publisher Full Text | Free Full Text

5. Goodman SN, Fanelli D, Ioannidis JPA: **What does research reproducibility mean?** *Sci Transl Med.* 2016; **8**(341): 341ps12.
PubMed Abstract | Publisher Full Text

6. Medley JK, Goldberg AP, Karr JR: **Guidelines for reproducibly building and simulating systems biology models.** *IEEE Trans Biomed Eng.* 2016; **63**(10): 2015–2020.
PubMed Abstract | Publisher Full Text | Free Full Text

7. Chen X, Dallmeier-Tiessen S, Dasler R, *et al.*: **Open is not enough.** *Nat Phys.* 2019; **15**(2): 113–119.
Publisher Full Text

8. R Core Team: **R: A Language and Environment for Statistical Computing**. R Foundation for Statistical Computing, Vienna, Austria. 2019.
Reference Source

9. King A, Nguyen D, Ionides E: **Statistical inference for partially observed markov processes via the R package pomp.** *J Stat Softw, Articles.* 2016; **69**(12): 1–43.
Publisher Full Text

10. Murray LM: **Bayesian State-Space modelling on High-Performance hardware using LibBi.** *J Stat Softw.* 2015; **067**(i10).
Publisher Full Text

11. van Deursen A, Klint P, Visser J: **Domain-specific languages: An annotated bibliography.** *ACM SIGPLAN Notices.* 2000; **35**(6): 26–36.
Publisher Full Text

12. Wickham H: **Advanced R**. CRC Press. 2014.
Publisher Full Text

13. Liu JS, Chen R: **Sequential monte carlo methods for dynamic systems.** *J Am Stat Assoc.* 1998; **93**(443): 1032–1044.
Publisher Full Text

14. Del Moral P: **Nonlinear filtering: Interacting particle resolution.** *Comptes Rendus de l'Académie des Sciences -Series I -Mathematics.* 1997; **325**(6): 653–658.
Reference Source

15. Arulampalam MS, Maskell S, Gordon N, *et al.*: **A tutorial on particle filters for online nonlinear/non-gaussian bayesian tracking.** *IEEE Trans Signal Process.* 2002; **50**(2): 174–188.
Reference Source

16. Schön TB, Svensson A, Murray L, *et al.*: **Probabilistic learning of nonlinear dynamical systems using sequential monte carlo.** *Mech Syst Signal Process.* 2018; **104**: 866–883.
Publisher Full Text

17. Endo A, van Leeuwen E, Baguelin M: **Introduction to particle markov-chain monte carlo for disease dynamics modellers.** *Epidemics.* 2019; **29**: 100363.
PubMed Abstract | Publisher Full Text

18. Ionides EL, Nguyen D, Atchadé Y, *et al.*: **Inference for dynamic and latent variable models via iterated, perturbed Bayes maps.** *Proc Natl Acad Sci U S A.* 2015; **112**(3): 719–24.
PubMed Abstract | Publisher Full Text | Free Full Text

19. Dagum L, Menon R: **OpenMP: An Industry-Standard API for Shared-Memory programming.** *IEEE Comput Sci Eng.* 1998; **5**(1): 46–55.
Publisher Full Text

20. Knock ES, Whittles LK, Lees JA, *et al.*: **The 2020 SARS-CoV-2 epidemic in England: key epidemiological drivers and impact of interventions.** *medRxiv.* 2021; 2021.01.11.21249564.
Publisher Full Text

21. Bauke H, Mertens S: **Pseudo random coins show more heads than tails.** *J Stat Phys.* 2004; **114**(3): 1149–1169.
Publisher Full Text

22. Bauke H, Mertens S: **Random numbers for large-scale distributed Monte Carlo simulations.** *Phys Rev E Stat Nonlin Soft Matter Phys.* 2007; **75**(6 Pt 2): 066701.
PubMed Abstract | Publisher Full Text

23. Blackman D, Vigna S: **Scrambled linear pseudorandom number generators**. 2018.
Reference Source

24. Matsumoto M, Nishimura T: **Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator.** *ACM Trans Model Comput Simul.* 1998; **8**(1): 3–30.
Publisher Full Text

25. Steele GL, Lea D, Flood CH: **Fast splittable pseudorandom number generators.** *SIGPLAN Not.* 2014; **49**(10): 453–472.
Publisher Full Text

26. Abadi M, Agarwal A, Barham P, *et al.*: **TensorFlow: Large-scale machine learning on heterogeneous systems**. 2015.
Reference Source

27. Box GEP, Muller ME: **A note on the generation of random normal deviates.** *Ann Math Stat.* 1958; **29**(2): 610–611.
Publisher Full Text

28. Devroye L: **Non-Uniform Random Variate Generation**. Springer, New York, NY. 1986.
Publisher Full Text

29. Hörmann W: **The generation of binomial random variates.** *J Stat Comput Sim.* 1993; **46**(1–2): 101–110.
Publisher Full Text

30. Knuth DE: **The art of computer programming,** volume 2 (3rd ed.): seminumerical algorithms. Addison-Wesley Longman Publishing Co., Inc., USA, 1997.
Reference Source

31. Hörmann W: **The transformed rejection method for generating poisson random variables.** *Insur Math Econ.* 1993; **12**(1): 39–45.
Publisher Full Text

32. Heston SL: **A Closed-Form solution for options with stochastic volatility with applications to bond and currency options.** *Rev Financ Stud.* 1993; **6**(2): 327–343.
Publisher Full Text

33. Doucet A, Johansen AM: **A tutorial on particle filtering and smoothing: Fifteen years later.** *Handbook of Nonlinear Filtering.* 2009; **12**: 01.
Reference Source

34. Kermack WO, McKendrick AG, Walker GT: **A contribution to the mathematical theory of epidemics.** *Proceedings of the Royal Society of London A.* 1927; **115**(772): 700–721.
Publisher Full Text

35. Anderson RM, Anderson B, May RM: **Infectious Diseases of Humans: Dynamics and Control**. OUP Oxford, 1992.
Reference Source

36. Allen EJ, Allen LJS, Arciniega A, *et al.*: **Construction of equivalent stochastic differential equation models.** *Stoch Anal Appl.* 2008; **26**(2): 274–297.
Publisher Full Text

37. Allen LJS: **A primer on stochastic epidemic models: Formulation, numerical simulation, and analysis.** *Infect Dis Model.* 2017; **2**(2): 128–142.
PubMed Abstract | Publisher Full Text | Free Full Text

38. Gillespie DT: **Approximate accelerated stochastic simulation of chemically reacting systems.** *J Chem Phys.* 2001; **115**(4): 1716–1733.
Reference Source

39. Plummer M, Best N, Cowles K, *et al.*: **CODA: convergence diagnosis and output analysis for MCMC.** *R News.* 2006; **6**(1): 7–11.
Reference Source

40. Roberts GO, Gelman A, Gilks WR: **Weak convergence and optimal scaling of random walk metropolis algorithms.** *Ann Appl Probab.* 1997; **7**(1): 110–120.
Publisher Full Text

41. Ionides EL, Breto C, Park J, *et al.*: **Monte Carlo profile confidence intervals for dynamic systems.** *J R Soc Interface.* 2017; **14**(132): 20170126.
PubMed Abstract | Publisher Full Text | Free Full Text

42. Ning N, Ionides E, Ritov Y: **Scalable Monte Carlo Inference and Rescaled Local Asymptotic Normality**. arXiv. 2020.
Reference Source

43. Doucet A, Pitt MK, Deligiannidis G, *et al.*: **Efficient implementation of Markov chain Monte Carlo when using an unbiased likelihood estimator.** *Biometrika.* 2015; **102**(2): 295–313.
Publisher Full Text

44. McElreath R: **Statistical rethinking: A Bayesian course with**

examples in R and Stan. CRC press. 2020.
Reference Source

45. Mossong J, Hens N, Jit M, *et al.*: **Social contacts and mixing patterns relevant to the spread of infectious diseases.** *PLoS Med.* 2008; **5**(3): e74.
PubMed Abstract | Publisher Full Text | Free Full Text

46. Plummer M: **JAGS: A program for analysis of bayesian graphical models using gibbs sampling**. In *Proceedings of the 3rd international workshop on distributed statistical computing.* 2003; **124**: 1–10.
Reference Source

47. Gelman A, Lee D, Guo J: **Stan: A probabilistic programming language for bayesian inference and optimization.** *J Educ Behav Stat.* 2015; **40**(5): 530–543.
Publisher Full Text

48. Andrade J, Duggan J: **An evaluation of hamiltonian monte carlo performance to calibrate age-structured compartmental SEIR models to incidence data.** *Epidemics.* 2020; **33**: 100415.
PubMed Abstract | Publisher Full Text

49. Toni T, Welch D, Strelkowa N, *et al.*: **Approximate Bayesian computation scheme for parameter inference and model selection in dynamical systems.** *J R Soc Interface.* 2009; **6**(31): 187–202.
PubMed Abstract | Publisher Full Text | Free Full Text

50. Fourment M, Claywell BC, Dinh V, *et al.*: **Effective online Bayesian phylogenetics via sequential Monte Carlo with guided proposals.** *Syst Biol.* 2018; **67**(3): 490–502.
PubMed Abstract | Publisher Full Text | Free Full Text

51. Doucet A, Godsill S, Andrieu C: **On sequential Monte Carlo sampling methods for Bayesian filtering.** *Stat Comput.* 2000; **10**: 197–208.
Publisher Full Text

52. L'Ecuyer P: **Good parameters and implementations for combined multiple recursive random number generators.** *Oper Res.* 1999; **47**(1): 1–173.
Publisher Full Text

53. L'ecuyer P, Simard R, Chen EJ, *et al.*: **An object-oriented random-number package with many long streams and substreams.** *Oper Res.* 2002; **50**(6): 923–1091.
Publisher Full Text

54. FitzJohn R: **mrc-ide/odin: v1.1.12** (Version v1.1.12). *Zenodo.* 2020.
http://www.doi.org/10.5281/zenodo.4772403

55. Lees J, FitzJohn R: **mrc-ide/odin.dust: v0.2.7** (Version v0.2.7). *Zenodo.* 2020.
http://www.doi.org/10.5281/zenodo.4772398

56. Lees J, FitzJohn R: **mrc-ide/dust: v0.9.3** (Version v0.9.3). *Zenodo.* 2020.
http://www.doi.org/10.5281/zenodo.4772395

57. Lees J, FitzJohn R: **mrc-ide/mcstate: v0.6.0** (Version v0.6.0). *Zenodo.* 2020.
http://www.doi.org/10.5281/zenodo.4772455

58. Lees J, FitzJohn R: **mrc-ide/odin-dust-plots: Plots at submission** (Version v1.0.0). *Zenodo.* 2020.
http://www.doi.org/10.5281/zenodo.4293396

Open Peer Review

Current Peer Review Status: ✓ ✓

Version 2

Reviewer Report 27 October 2021

https://doi.org/10.21956/wellcomeopenres.18728.r44387

✓

Edward Ionides
Department of Statistics, University of Michigan, Ann Arbor, MI, USA

I've looked through the revision, which is thorough. Please update the status of my recommendation to "Approved". I did notice many capitalization errors in the references - perhaps an artifact of using BibTex? I recommend these are fixed, though it is not scientifically important."

Competing Interests: No competing interests were disclosed.

I confirm that I have read this submission and believe that I have an appropriate level of expertise to confirm that it is of an acceptable scientific standard.

Version 1

Reviewer Report 26 January 2021

https://doi.org/10.21956/wellcomeopenres.18131.r41899

✓

Rene Niehus
Center for Communicable Disease Dynamics, Department of Epidemiology, Harvard T H Chan School of Public Health, Boston, MA, USA

Knock *et al*. have developed a software that allows building and fitting of state space models to (partially) observed data and that allows simulation from those models. Like tools such as WinBugs or Stan their tool allows the user to define models in language that is similar to statistical modelling language. It is efficient in fitting time-series via transcribing the model into fast C++ language, parallelisation, and the use of particle filtering, similar to pomp or libBo. It integrates well with commonly used R environment, and it is thus available to use for a wide group of researchers.

I would not be surprised if this software and its iterations took an important role for predicting and understanding the ongoing COVID-19 pandemic. A key determinant for this is its development and refinement by an interdisciplinary team, and working with feedback from use case in an urgent public health crisis.

I truly enjoyed reading this paper, and I would like to suggest a few points of improvement, before I make some suggestions for improved readability.

1. The authors describe at first some general concepts, then the different components of their framework, the random number generator, its operation, and finally a comparison with other tools. It should be made clearer at the very beginning, in what ways this software advances current tools, and combines the advantages of different tools. It remains unclear if the dust-object idea containing particles is novel, or an adaptation. I also suggest extending Table 1 to include several of the other features that differ between dust, pomp, and libBi.

2. The tool of the authors follows a Bayesian framework, which I consider especially suitable in the context of COVID-19 modelling where information on biological parameters is added constantly through various trials. I am not entirely sure, but I think the Figure 3 shows a prior-predictive simulations/checks: Once a user has written the structure of a model and they want to understand the implications of the prior choice on the outcome-scale, then they would want to simulate from the model and its priors (instead of posteriors)(see McElreath 2020, Chapter 4.3.2). As this is one of the strengths of this simulation-based tool, it would be useful if the authors added prior predictive checks as an explicit step in the described work flow.

3. Following the above point, I am wondering if the user is limited to priors in the shape of standard distributions (e.g., Normal(0,1)), or would it be possible for the user to define a prior based on a posterior from a different model, for example in the form of 1000 random draws of the posterior? While I see how this is a computational challenge, I think it is a feature that might be very useful for adding results from other Bayesian studies. Can the authors at least comment on this?

4. Is it possible to include time varying variables, such as a R0 that is changing in time? It is not clear from the text if this is possible. If yes, a reader might benefit from advise on how to smooth such a variable over time to avoid overfitting.

5. It would be very helpful to have a cartoon that visualises how odi/dust/mcstate work together and how they make use of the abstractions (particle and dust) and how all this system generates the model building pipeline.

As this paper is meant to help researchers like for example infectious disease modellers to make use of sophisticated software, it will be useful to revise the text to help this target audience to better follow the paper. Here are some concrete ideas:

- A non-technical reader will not gain much from Figure 1 without further comparisons or explanation. What does a straight line signify? Should this surprise, what is to expect?

- Briefly explain what a dynamics library is.
- Briefly explain why C++ is used at the backend (speed).
- Briefly explain (or avoid) the term transpile.
- “change in randomness” page 6 should be explained more precisely.
- Redefine what “m” mean on page 6.
- The indexing in the expression m_{i} seems odd if m is defined as in integer above. Please explain or change.
- Is the variance-covariance matrix for the proposal kernel computed in an automated step, or does the user have to do this, or make that setting?
- Adding example code that uses the predict() function would help.
- “adding square brackets to the left hand side of each declaration”, a reference to a code block should be added. It would help to enumerate the code blocks for easy reference.
- It is not clear how entries of the contact matrix m are defined
- Equation (1) needs definition of beta, N, I and m.

I found a few typos
“psuedorandom numbers” page 4
“first dimension[s]” page 12
“and and” page 15

References
1. McElreath R: Statistical rethinking: A Bayesian course with examples in R and Stan. *CRC press*. Mar 2020.

Is the rationale for developing the new software tool clearly explained?
Partly

Is the description of the software tool technically sound?
Yes

Are sufficient details of the code, methods and analysis (if applicable) provided to allow replication of the software development and its use by others?
Yes

Is sufficient information provided to allow interpretation of the expected output datasets and any results generated using the tool?

Yes

Are the conclusions about the tool and its performance adequately supported by the findings presented in the article?
Yes

Competing Interests: No competing interests were disclosed.

Reviewer Expertise: Infectious disease epidemiology, COVID-19 dynamics, microbiome dynamics.

I confirm that I have read this submission and believe that I have an appropriate level of expertise to confirm that it is of an acceptable scientific standard.

Author Response 03 Jun 2021

John Lees, Imperial College London, UK

Knock et al. have developed a software that allows building and fitting of state space models to (partially) observed data and that allows simulation from those models. Like tools such as WinBugs or Stan their tool allows the user to define models in language that is similar to statistical modelling language. It is efficient in fitting time-series via transcribing the model into fast C++ language, parallelisation, and the use of particle filtering, similar to pomp or libBo. It integrates well with commonly used R environment, and it is thus available to use for a wide group of researchers.

I would not be surprised if this software and its iterations took an important role for predicting and understanding the ongoing COVID-19 pandemic. A key determinant for this is its development and refinement by an interdisciplinary team, and working with feedback from use case in an urgent public health crisis.

I truly enjoyed reading this paper, and I would like to suggest a few points of improvement, before I make some suggestions for improved readability.

We thank the reviewer for their kind comments and positive summary. We have a number of additional changes to the text which we believe should clarify the points arising.

The authors describe at first some general concepts, then the different components of their framework, the random number generator, its operation, and finally a comparison with other tools. It should be made clearer at the very beginning, in what ways this software advances current tools, and combines the advantages of different tools. It remains unclear if the dust-object idea containing particles is novel, or an adaptation. I also suggest extending Table 1 to include several of the other features that differ between dust, pomp, and libBi.

We are somewhat constrained by the software manuscript format: demonstrating motivation and specific use cases before a full comparison with other methods – so we have left the full comparison with the alternative packages before the summary. However, we do agree that this would be useful to be introduced earlier on, so have added more text to the introduction to clarify the differences between packages, and expanded table 1 as suggested.

We are not sure of the exact manner in which pomp and libBi are engineering in this regard, but this kind of class definition is a fairly typical occurrence in C++ libraries. But, we have added more of a description of why we opted for the Dust/Particle separation, which we think makes the rationale for this decision clearer, and was missing previously. We have also added and reference a “design” vignette, which describes this in further detail.

The tool of the authors follows a Bayesian framework, which I consider especially suitable in the context of COVID-19 modelling where information on biological parameters is added constantly through various trials. I am not entirely sure, but I think the Figure 3 shows a prior-predictive simulations/checks: Once a user has written the structure of a model and they want to understand the implications of the prior choice on the outcome-scale, then they would want to simulate from the model and its priors (instead of posteriors)(see McElreath 2020, Chapter 4.3.2). As this is one of the strengths of this simulation-based tool, it would be useful if the authors added prior predictive checks as an explicit step in the described work flow.

We now follow the reviewer’s suggestion, and add a section on simulating from the priors to the workflow. Notably, we have added a simulate method to the dust package which makes running models across the time series using a specific set of parameters easier and more flexible.

Following the above point, I am wondering if the user is limited to priors in the shape of standard distributions (e.g., Normal(0,1)), or would it be possible for the user to define a prior based on a posterior from a different model, for example in the form of 1000 random draws of the posterior? While I see how this is a computational challenge, I think it is a feature that might be very useful for adding results from other Bayesian studies. Can the authors at least comment on this?

One advantage of our framework’s tight integration with R is that functions for the likelihood and prior are completely flexible, and can use any functionality the R language allows. Although we demonstrate with a simple prior distribution in the use cases, any function can be defined. This can include using functions from external packages, or, as suggested, drawing from another model’s posterior. We’ve noted this more specifically in the revised text. A future feature will include the ability to ‘restart’ models from part way along the time series, and will draw on this suggestion to re-initialise the model.

Is it possible to include time varying variables, such as a R0 that is changing in time? It is not clear from the text if this is possible. If yes, a reader might benefit from advise on how to smooth such a variable over time to avoid overfitting.

Indeed, this is possible, and used extensively in the SIRCOVID package. To do so in these packages usually requires adding a simple ‘transform’ function in the particle filter, which translates between the parameters being inferred in R, and parameters in the model code. The easiest way to implement time-varying parameters is to make them piecewise-linear, and point the reader to the example in SIRCOVID of how to set this up (but do add a paragraph noting this to the examples). In theory a smoothing method such a spline fitting using a few free parameters would also be possible, but we do not demonstrate that here.

It would be very helpful to have a cartoon that visualises how odin/dust/mcstate work together and how they make use of the abstractions (particle and dust) and how all this system generates the model building pipeline.

We have added an extra figure giving an overview of the software packages.

As this paper is meant to help researchers like for example infectious disease modellers to make use of sophisticated software, it will be useful to revise the text to help this target audience to better follow the paper. Here are some concrete ideas:

Thank you for these suggestions, which we address individually below. We expect that many modellers will find the package vignettes particularly useful in addition to this paper – there we are able to separate out technical language on design decisions with practical guides on how to use the software.

A non-technical reader will not gain much from Figure 1 without further comparisons or explanation. What does a straight line signify? Should this surprise, what is to expect?

Added to caption.

Briefly explain what a dynamics library is.

Added.

Briefly explain why C++ is used at the backend (speed).

Added.

Briefly explain (or avoid) the term transpile.

Added.

“change in randomness” page 6 should be explained more precisely.

Explanation added.

Redefine what “m” mean on page 6.

The indexing in the expression m_{i} seems odd if m is defined as in integer above. Please explain or change.

Thanks for spotting this – this should have been *p*, indexed for each core/thread not chain.

Is the variance-covariance matrix for the proposal kernel computed in an automated step, or does the user have to do this, or make that setting?

This is indeed a user option. We note that in the text, and a package vignette discusses how to set this in more detail than we have space for in this paper.

Adding example code that uses the predict() function would help.

Added.

“adding square brackets to the left hand side of each declaration”, a reference to a code block should be added. It would help to enumerate the code blocks for easy reference.

Reference to code block added. We are unfortunately unable to give line numbers to the code blocks due to formatting constraints.

It is not clear how entries of the contact matrix m are defined

We have reordered the text slightly, and expanded the description.

Equation (1) needs definition of beta, N, I and m.
I found a few typos
“psuedorandom numbers” page 4
“first dimension[s]” page 12
“and and” page 15

Typos fixed.

Competing Interests: No competing interests were disclosed.

Reviewer Report 20 January 2021

https://doi.org/10.21956/wellcomeopenres.18131.r42058

? **Edward Ionides**
Department of Statistics, University of Michigan, Ann Arbor, MI, USA

The authors have built a framework for model development and statistical inference for complex partially observed stochastic dynamic models, motivated by applications in epidemiology. The methods are simulation-based, offering scientists appealing flexibility to investigate a range of different model specifications. The resulting odin/dust/mcstate software environment is compared with two other packages having similar capabilities, libBi and pomp. Time comparisons are comparable, but odin/dust/mcstate has had considerable work put into some important practical considerations: (i) the domain-specific language (DSL) used for model specification; (ii) parallelization issues.

The work undertaken by the authors contributes a new perspective to the worthwhile task of building and testing models to interpret data on noisy and incompletely measured dynamic systems. I shall compare odin/dust/mcstate to my experiences with pomp (King *et al*, 2016)[1] raising some points that are missing from the current version of the authors' manuscript.

1. The authors emphasize only Bayesian inference. The Bayesian paradigm is convenient for combining uncertainty about parameters with variability inherent in the dynamics, making it well placed to provide forecasts once the prior and model are satisfactory. However, in some situations there are advantages to avoiding the additional requirement to add to the model a quantification of prior uncertainty about parameters. For model criticism, the goal is to diagnose features of the model that are incompatible with the data. If such features exist, then a compatible prior does not exist and so the requirement to specify one is a hindrance rather than a help. It may be better to first investigate model specification by cycles of the scientific method promoted by Popper (1959)[2]: What features of the data are inconsistent with the hypothesized model? Can we find a better model?

2. Maximum likelihood estimation, and associated likelihood ratio tests and profile likelihood confidence intervals, provide non-Bayesian methodology for partially observed stochastic dynamic models that also make statistically efficient use of data. The odin/dust/mcstate framework could, for example, implement an iterated filtering likelihood maximization (Ionides *et al*, 2015)[3] as has been much used in pomp.

3. Monte Carlo adjusted profile likelihood estimation (Ionides *et al*, 2017)[4] has some favorable scaling properties (Ning *et al*, 2020)[5] that are not shared by particle Markov chain Monte Carlo (Doucet *et al*, 2015)[6]. For analysis that stretches available computational resources, computational scaling may be another reason to consider likelihood-based inference.

4. I agree with the authors on the importance of developing software accessible to a broad technical audience. The DSL written by the authors builds on the success of packages such as WinBUGS, JAGS, and stan. The Csnippet approach used by pomp is somewhat different: it gives the users the full flexibility of C while hiding certain details such as variable definitions for latent states, observations and parameters. For simple models, pomp Csnippet code can look very much like WinBUGS DSL code. For more complex models, the user may start needing to use increasingly esoteric properties of C, but this is a strength as well as a weakness, since it is hard to anticipate and encode in a DSL all the features one might want in a simulation model. The Csnippet framework is simple enough that it is accessible to students in a Masters level time series course where students with no prior C experience develop and fit their own pomp analysis.

5. The benefits of parallelizing the particle filter depend on the purpose to which it is put. If it is used in a single, long, particle Monte Carlo Markov chain calculation then parallelization may help. However, in practice, one should carry out many replications of the inferential algorithm while varying the starting point and the seed of the random number generator. In a likelihood-based framework, one may want to evaluate and maximize the likelihood at a range of profile points. In these cases, parallelizing within the particle filter can be inferior to parallelizing over the embarassingly parallel replicated filters.

References
1. King A, Nguyen D, Ionides E: Statistical Inference for Partially Observed Markov Processes via theR Packagepomp. *Journal of Statistical Software*. 2016; **69** (12). Publisher Full Text
2. Popper K: The Logic of Scientific Discovery. *Hutchinson*. 1959.
3. Ionides EL, Nguyen D, Atchadé Y, Stoev S, et al.: Inference for dynamic and latent variable models via iterated, perturbed Bayes maps.*Proc Natl Acad Sci U S A*. 2015; **112** (3): 719-24 PubMed Abstract | Publisher Full Text
4. Ionides EL, Breto C, Park J, Smith RA, et al.: Monte Carlo profile confidence intervals for dynamic systems.*J R Soc Interface*. **14** (132). PubMed Abstract | Publisher Full Text
5. Ning N, Ionides E, Ritov Y: Scalable Monte Carlo Inference and Rescaled Local Asymptotic Normality. *arXiv*. 2020. Reference Source
6. Doucet A, Pitt M, Deligiannidis G, Kohn R: Efficient implementation of Markov chain Monte Carlo when using an unbiased likelihood estimator. *Biometrika*. 2015; **102** (2): 295-313 Publisher Full Text

Is the rationale for developing the new software tool clearly explained?
Yes

Is the description of the software tool technically sound?
Yes

Are sufficient details of the code, methods and analysis (if applicable) provided to allow replication of the software development and its use by others?
Yes

Is sufficient information provided to allow interpretation of the expected output datasets and any results generated using the tool?
Yes

Are the conclusions about the tool and its performance adequately supported by the findings presented in the article?
Yes

Competing Interests: No competing interests were disclosed.

Reviewer Expertise: Time series analysis, with applications in epidemiology and ecology.

I confirm that I have read this submission and believe that I have an appropriate level of expertise to confirm that it is of an acceptable scientific standard, however I have significant reservations, as outlined above.

Author Response 03 Jun 2021

John Lees, Imperial College London, UK

The authors have built a framework for model development and statistical inference for complex partially observed stochastic dynamic models, motivated by applications in epidemiology. The *methods are simulation-based, offering scientists appealing flexibility to investigate a range of different model specifications. The resulting odin/dust/mcstate software environment is compared with two other packages having similar capabilities, libBi and pomp. Time comparisons are comparable, but odin/dust/mcstate has had considerable work put into some important practical considerations: (i) the domain-specific language (DSL) used for model specification; (ii) parallelization issues.*

The work undertaken by the authors contributes a new perspective to the worthwhile task of building and testing models to interpret data on noisy and incompletely measured dynamic systems. I shall compare odin/dust/mcstate to my experiences with pomp (King et al, 2016)[1] raising some points that are missing from the current version of the authors' manuscript.

We thank the reviewer for their comments and thoughtful suggestions. We have made two major additions to the code to add the functionality as suggested, and modified the text to better address the issues raised. We have also added some of this summary into our introduction, to introduce pomp and libBi earlier on.

The authors emphasize only Bayesian inference. The Bayesian paradigm is convenient for combining uncertainty about parameters with variability inherent in the dynamics, making it well placed to provide forecasts once the prior and model are satisfactory. However, in some situations there are advantages to avoiding the additional requirement to add to the model a quantification of prior uncertainty about parameters. For model criticism, the goal is to diagnose features of the model that are incompatible with the data. If such features exist, then a compatible prior does not exist and so the requirement to specify one is a hindrance rather than a help. It may be better to first investigate model specification by cycles of the scientific method promoted by Popper (1959)[2]: What features of the data are inconsistent with the hypothesized model? Can we find a better model?

It is true that we focused on Bayesian inference using a prior and model which have been determined to be satisfactory, which leaves out the model criticism phase. We have now added two sections on model criticism, alternatives to prior specification, and maximum likelihood estimation (noting the additions in response to the second point). We hope this expands to scope of discussion here suitably.

Maximum likelihood estimation, and associated likelihood ratio tests and profile likelihood confidence intervals, provide non-Bayesian methodology for partially observed stochastic dynamic models that also make statistically efficient use of data. The odin/dust/mcstate framework could, for example, implement an iterated filtering likelihood maximization (Ionides et al, 2015)[3] as has been much used in pomp.

Thank you for this suggestion, we agree that this would be a nice addition, and fits well with the design of our packages. We have now implemented and tested this algorithm using dust and mcstate. We have also added a package vignette and section to this paper describing its use.

Monte Carlo adjusted profile likelihood estimation (Ionides et al, 2017)[4] has some favorable scaling properties (Ning et al, 2020)[5] that are not shared by particle Markov chain Monte Carlo

(Doucet et al, 2015)^6. For analysis that stretches available computational resources, computational scaling may be another reason to consider likelihood-based inference.

We hope that our additions to the text and the addition of the iterated filtering algorithm now allow likelihood-based inference in our packages. Although we have not added the MCAP algorithm to the package code, we have added this point and associated references to the text, as is done in pomp.

I agree with the authors on the importance of developing software accessible to a broad technical audience. The DSL written by the authors builds on the success of packages such as WinBUGS, JAGS, and stan. The Csnippet approach used by pomp is somewhat different: it gives the users the full flexibility of C while hiding certain details such as variable definitions for latent states, observations and parameters. For simple models, pomp Csnippet code can look very much like WinBUGS DSL code. For more complex models, the user may start needing to use increasingly esoteric properties of C, but this is a strength as well as a weakness, since it is hard to anticipate and encode in a DSL all the features one might want in a simulation model. The Csnippet framework is simple enough that it is accessible to students in a Masters level time series course where students with no prior C experience develop and fit their own pomp analysis.

Thank you for this perspective on the Csnippet approach. We have expanded our discussion of the DSL to note the strengths and weaknesses of either approach from a user point of view.

If greater flexibility is required, we also note that it is possible to directly write C/C++ code to be used as a dust model (and therefore with the rest of our framework). The easiest way of doing this is by modifying one of our examples – by just changing single sections of the model definition this ends up being somewhat similar to the Csnippet approach. For testing some models and new features of dust, this is the route we took. Our experience with real-life complex models has been that the odin DSL was greatly preferred due to its built-in bookkeeping of model indices, and did not limit any of the required features.

As a further alternative, we have also added the ability to the odin DSL to call arbitrary C++ functions – essentially imitating the Csnippet approach where the DSL may prove limiting.

The benefits of parallelizing the particle filter depend on the purpose to which it is put. If it is used in a single, long, particle Monte Carlo Markov chain calculation then parallelization may help. However, in practice, one should carry out many replications of the inferential algorithm while varying the starting point and the seed of the random number generator. In a likelihood-based framework, one may want to evaluate and maximize the likelihood at a range of profile points. In these cases, parallelizing within the particle filter can be inferior to parallelizing over the embarassingly parallel replicated filters.

This is an excellent point, and as such we have now expanded the flexibility of the parallelisation. Users can now parallelise both over particles and over filters independently, and choose how to do so depending on their problem. The interface is straightforward – between chain parallelism is enabled by increasing the number of workers, and specifying a total number of threads available.

The issue of varying the starting point is addressed by the 'initial' argument to the particle filter, through which the user can specify an arbitrary R function to generate the initial conditions for the filter.

We have added the pmcmc_chains_prepare() function to address the issue of varying the starting seed. This uses the 'long jump' feature of the Xoshiro random number generator to generate streams which are uncorrelated even when further parallelisation is used within them. This function automates both the process of initialising a set of chains with different (and uncorrelated) seeds and initial conditions.

We concur that parallelising within the particle filter can be inferior to parallelising chains – in some cases we have observed up to a 40% loss of efficiency (mostly due to particle divergence). We have found the added flexibility on this matter of great practical use in large problems on large distributed systems. Using code to orchestrate and collect information from independent runs, we can now run independent particle filters on separate computer nodes, and nested within these parallelise further over particles on each shared memory node. This top level of parallelisation can also be used to run with different parameters, or using the same model with different data (for example, fitting a COVID-19 model to different regions).

Competing Interests: No competing interests were disclosed.

Comments on this article

Version 2

Reader Comment 11 Aug 2021

Penny Hancock, University of Oxford, UK

Dear Authors, please see below my review of this paper. Although it could not be published due to my new Imperial affiliation I am providing it here in case it is helpful.
Best wishes, Penelope Hancock

Fitzjohn *et al*. present a series of R packages that provide computational functionality for simulating and fitting state space models, with consideration given to computational efficiency and parallelization, and principled random number generation, while allowing flexible model design and model checking. It is very encouraging to see such advances in the computational aspects of state space model development, particularly aimed at epidemiological modellers, allowing accessible, reproducible and efficient implementation of intense computation for Bayesian and ML models. The methodology is timely and well thought out. On reading the ms, I did encounter a few issues that I would like to see addressed:

- When reading through the introduction, it is not clear to me what is meant by ‘stochastic model’, as there are a number of ways that epidemiological modellers might implement stochasticity. It might be helpful to cover the different model types mentioned on page 16 under the section ‘Stochastic SIR model’ in the Introduction in order to clarify the main types of stochastic models that are supported. I find it odd that the first use case that is presented is a financial model, given that the package is “targeted at infectious disease modellers but suitable for other domain users”. Readers of this paper are likely to skim quickly to the ‘use case’ section, and could be put off by this. Perhaps the ODE and CTMC models should be presented first, and the stochastic ODE case mentioned later (with epidemiological and non-epidemiological examples)? I think it is also worth a greater emphasis on the applicability to non-stochastic ODEs, which are very common in epidemiological modelling. On page 11 it is stated that SDEs ‘better represent the variance in real world systems’, however it is often difficult to find data to parameterise the form of that variance. I think this statement should be supported with evidence and citations, and I would avoid the implication that stochastic models are always better.

- I think the limitations of the methodology need to be more clearly explicated. On page 16 there is a brief passing mention of applicability of the method to spatial models, and delay differential models, but this is not demonstrated or clarified by any statements about limitations of the method. For example, parameter inference on spatial models using MCMC methods is often challenging due to high correlations across inferred parameters. Methods such as the Hamiltonian MCMC implemented in STAN will perform better in such situations than standard Metropolis Hastings. I think a few sentences about the limitations, and avenues for future development, of this package would be helpful to aid users in selecting appropriate tools.

Other comments:

- page 13. The cases_compare function is not very clear to me. Variables like incidence_observed are not defined.

- page 14. ‘the observations must be evenly spaced’ – many epi data sets not likely to meet this requirement?

- page 14. ‘the right panel of Figure 5’ – Figure 5 has 4 panels, please refer to the letter of the panel.

- page 16. great to see convergence diagnostics and prior predictive checks!

Competing Interests: No competing interests were disclosed.