Journal of Statistical Software

November 2016, Volume 75, Issue 2. doi: 10.18637/jss.v075.i02

CollocInfer: Collocation Inference in Differential Equation Models

Giles Hooker
Cornell University

James O. Ramsay
McGill University

Luo Xiao
North Carolina State University

Abstract

This monograph details the implementation and use of the **CollocInfer** package in R for smoothing-based estimation of continuous-time nonlinear dynamic systems. These routines represent an extension of the *generalized profiling* methods in Ramsay, Hooker, Campbell, and Cao (2007) for estimating parameters in nonlinear ordinary differential equations. An interface to the **fda** package is included. The package also supports discrete-time systems. We describe the methodological and computational framework and the necessary steps to use the software. Equivalent functionality is available in MATLAB.

Keywords: differential equation, collocation, profiled estimation, smoothing.

1. Introduction

1.1. Background, notation and overview

This paper details the **CollocInfer** (Hooker, Xiao, and Ramsay 2016) package for using basis expansion methods to estimate parameters in differential equation models in both the R (R Core Team 2016) and MATLAB (The MathWorks, Inc. 2011) programming languages. The **CollocInfer** package supports the generalized profiling (GP) methods in Ramsay *et al.* (2007). It also provides some diagnostic plots and testing procedures, as well as access to gradient matching (GM) methods. The **CollocInfer** package automatically installs and works with the functional data analysis (FDA) package **fda** (Ramsay, Wickham, Graves, and Hooker 2014). In this paper, we will use squared-error criteria to fit data; however more general criteria is also accommodated by the package.

We assume that the reader is familiar with the basic theory of ordinary differential equations (ODEs) as well as with basis expansion systems and nonparametric smoothing. We will

therefore give these topics only a cursory introduction; the reader requiring more background is directed to Ramsay and Silverman (2005) or Ramsay, Hooker, and Graves (2009).

All of the example analyses in this paper are presented in R code. However, code in the MATLABs language is available as well in the **CollocInfer** package and is structured to look as much as possible like the R code. We provide some comments on the differences between the languages in Section 7; however, we think MATLAB users can easily translate the R examples into MATLAB.

A typical differential equation model involves one or several functions, which we designate by $x_j, j = 1, \ldots, d$. The literature on dynamical systems often refers to these functions as *states*, and to values of functions over a range of time values t as *trajectories*. We use the notation Dx for the first derivative of function x rather than the more classic notation dx/dt; and $Dx(t)$ for the value of the derivative at time t. Higher order derivatives whenever needed are indicated by $D^2x, D^3x, \ldots$, here we note that high order ordinary differential can always be re-expressed as systems of first order differential equations.

We begin with the problem set-up. The system under study is assumed to be governed by the model

$$\begin{aligned} Dx_1(t) &= f_1(x_1(t), \ldots, x_d(t); t, \theta) \\ &\vdots \\ Dx_j(t) &= f_j(x_1(t), \ldots, x_d(t); t, \theta) \\ &\vdots \\ Dx_d(t) &= f_d(x_1(t), \ldots, x_d(t); t, \theta) \end{aligned} \tag{1}$$

or, in matrix notation,

$$D\mathbf{x}(t) = \mathbf{f}(\mathbf{x}(t); t, \theta). \tag{2}$$

Here $\mathbf{x}$ is a d-dimensional *state vector* of functions that describes the system as it changes over time. The vector $\mathbf{f}$ of d functions f_j maps $\mathbf{x}$ to its first derivative $D\mathbf{x}$, and this map may depend on t directly. Map $\mathbf{f}$ also depends on a p-dimensional parameter vector θ.

We assume that the system is measured at times $t_1, \ldots, t_n$, which by default are assumed to be common to all observed states. The values of the measurements are a set of n vectors $\mathbf{y}_1, \ldots, \mathbf{y}_n$, each of size d, and from the measurements we wish to obtain an estimate of the parameter vector θ. The measurements are subject to additive measurement errors

$$\mathbf{y}_i = \mathbf{x}(t_i) + \boldsymbol{\epsilon}_i,$$

where $\boldsymbol{\epsilon}_i = (\epsilon_{i1}, \ldots, \epsilon_{id})^\top$ and the errors ϵ_{ij} are assumed to have Gaussian distributions that are independent between time points and independent of the values of $\mathbf{x}$. The data vectors $\mathbf{y}_i$ are allowed to have missing measurements, represented as `NA` in R. Usually only a subset of the d states x_j are measured; for those unmeasured states we provide `NA` values for the corresponding entries in $\mathbf{y}_i$. **CollocInfer** can also account for indirect measurements of states, the accommodation of which will be detailed in Section 4.2.

The methods described here are the generalized profiling methods in Ramsay *et al.* (2007) and to our knowledge **CollocInfer** provides the only implementation of these methods – although it builds on the earlier MATLAB software released at the time of the paper. Relevant stochastic (i.e., probabilistically-evolving) methods in R can be found in the packages **pomp**

(King, Nguyen, and Ionides 2016) and **smfsb** (Wilkinson 2013) using maximum likelihood or in **sl** (Wood 2010) using synthetic likelihood. In other software environments, the **System Identification** toolbox in MATLAB (see Ljung 1995) can be employed for some systems. The **IPOPT** package (Wächter and Biegler 2006) utilizes basis expansions and works with **AMPL** (Fourer, Gay, and Kernighan 2003); a detailed description of the parameter estimation methods is available in Tjoa and Biegler (1991) and Hooker and Biegler (2007). Similar ideas based on Runge-Kutta methods are implemented in the **VPLAN** package (Körkel 2002); see the multiple shooting methods in Bock (1983) for details. The **DEDiscover** software described in Wu, Miao, Warnes, Wu, LeBlanc, Dykes, and Demeter (2008) also provides tools for exploring differential equation solutions and parameter estimation. However, these all assume that either the ODE provides an exact description of the trajectory of the system, or that an explicit probabilistic description of the system's dynamics is available. **CollocInfer** allows for additional flexibility in assuming that the ODE's description of the dynamics are only approximately correct, but without requiring an explicitly stochastic model of system evolution.

The rest of this section provides background material on the numerical methods employed by **CollocInfer** particularly with reference to methods in packages **deSolve** and **lsoda** (Soetaert, Petzoldt, and Setzer 2010). Section 1.2 describes the FitzHugh-Nagumo equations as a specific example of ODEs and some functionality for working with them in R, while Section 1.3 details the basis functions that we will use to approximate trajectories $\mathbf{x}(t)$ and 1.4 will illustrate their use with spline smoothing as an initial step. With these tools, Section 2 will present methods of parameter estimation, and in particular the gradient matching and profiling methods implemented in **CollocInfer**. Section 3 will describe how to apply these methods using **CollocInfer**, Section 4 illustrates some already-implemented extensions with a model from experimental ecology, and Section 5 discusses further ways in which **CollocInfer**'s functionality can be extended. An example of this extensibility is given in 6 which illustrates the application of these methods to discrete-time dynamics. Appendices discuss equivalent functionality in MATLAB as well as providing technical details.

1.2. An example ODE: The FitzHugh-Nagumo spike potential equations

ODEs are not commonly part of a standard statistics curriculum and we this subsection is intended to serve as a brief exposition of the class of models for which **CollocInfer** is designed as well as presenting functionality in R – particularly in the package **deSolve** (Soetaert *et al.* 2010) – for solving them. In particular, we will illustrate these methods using the FitzHugh-Nagumo equations (FitzHugh 1961; Nagumo, Arimoto, and Yoshizawa 1962; see also Wilson 1999), which will be employed to demonstrate **CollocInfer** later. The equations describe a two-state system, which is relatively simple but can illustrate many practical aspects of fitting differential equation models to data. We provide code to generate and analyze data as we go along. Code to run this analysis can also be found in the `fhn` demo in the package.

Neurons communicate by sending "pulses" of voltage down their axons to other neurons. These pulses can be described by the FitzHugh-Nagumo equations, a simplification of the Hodgkins-Huxley equations (Hodgkin and Huxley 1952). The equations are

$$\begin{aligned} DV &= c(V - V^3/3 + R) \\ DR &= (V - a + bR)/c, \end{aligned} \tag{3}$$

where V is the voltage. The pulse is generated by the interplay of several chemical ions that

cross the membrane boundary and R (for "recovery") represents the combined flow of two of these that act to counter the build-up of voltage. Typically R cannot be measured, but we will start off assuming that both are measured. The parameters (a, b, c) in the equations are left unspecified and need to be estimated.

Equations 3 uniquely define functions $V(t)$ and $R(t)$ so long as the initial conditions or states $V(t_0) = V_0$ and $R(t_0) = R_0$ are known. However, solutions $(V(t), R(t))$ to (3) cannot be written down algebraically and have to be approximated by numerical methods. Often these solutions are approximated by Runge-Kutta or linear multistep methods, which, in R, can be carried out by functions in the package **deSolve**, such as `lsoda`. See Deuflhard and Bornemann (2000) for an overview of numerical methods for solving ODEs.

Before we begin, we define a couple of vectors of names for the indices of arrays containing parameters and values of the trajectories. These are

```
R> FhNvarnames <- c("V", "R")
R> FhNparnames <- c("a", "b", "c")
```

To use `lsoda`, we specify the initial conditions (the values of V and R at time 0) to be, respectively,

```
R> x0 <- c(-1, 1)
R> names(x0) <- FhNvarnames
```

In the above we have added names to the elements of `x0` so as to keep track of them easily. We also need initial values for the parameter vector $\theta = (a, b, c)$,

```
R> FhNpars <- c(0.2, 0.2, 3)
R> names(FhNpars) <- FhNparnames
```

Next, we need a function to evaluate the FitzHugh-Nagumo equations. The **deSolve** package requires that such a function will return a named list, with a member named `fn` and containing the function evaluation

```
R> fhn.ode <- function(times, x, p) {
+    dx <- x
+    dimnames(dx) <- dimnames(x)
+    dx["V"] <- p["c"] * (x["V"] - x["V"]^3 / 3 + x["R"])
+    dx["R"] <- - (x["V"] - p["a"] + p["b"] * x["R"]) / p["c"]
+    return(list(dx))
+  }
```

Finally, we define 401 equally spaced time values

```
R> FhNplottimes <- seq(0, 20, 0.05)
```

We can now call function `lsoda` to compute the approximation to the solution at these time values

```
R> out <- lsoda(x0, times = FhNplottimes, fhn.ode, FhNpars)
```

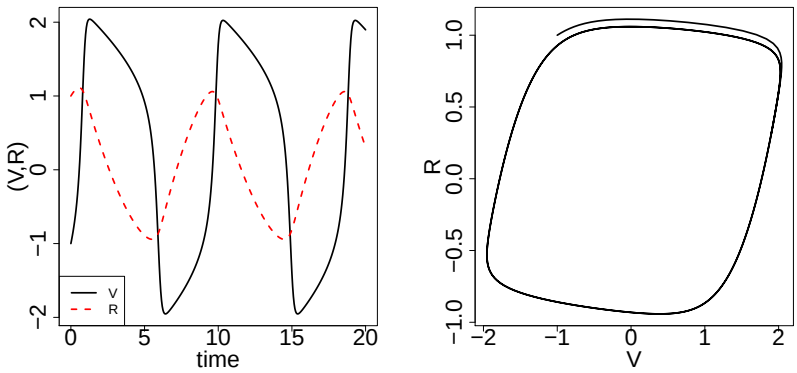

Figure 1: Solutions to the FitzHugh-Nagumo Equations (3). Left: V and R plotted against time. Right: R plotted against V.

The output of `lsoda` is an $n \times 3$ matrix where the first column has the evaluation times. If we plot these solutions against time we obtain a wave

```
R> matplot(out[, 1], out[, 2:3], type = "l")
R> legend("bottomleft", c("V", "R"))
```

Or, on the phase space (V versus R) plot, we obtain a square-ish orbit

```
R> plot(out[, 2], out[, 3], type = "l", xlab = "V", ylab = "R")
```

The plots by these commands are given in Figure 1.

We now generate some data at $n = 41$ equally spaced points, simulating independent measurement errors from a normal distribution with mean zero and standard deviation 0.1. To generate data we re-run `lsoda` add noise:

```
R> FhNtimes <- seq(0, 20, 0.5)
R> FhNn <- length(FhNtimes)
R> out <- lsoda(x0, times = FhNtimes, fhn.ode, FhNpars)
R> FhNdata <- y[, 2:3] + 0.1 * matrix(rnorm(2 * FhNn), FhNn, 2)
```

The specific data employed to exactly reproduce the results from our analysis can be obtained from `data("FhNdata")`, but the data generated by the above commands should produce something very similar.

1.3. Basis functions for representing trajectories

This subsection presents basis expansions as the numerical tool underlying **CollocInfer** which will be used as an means or representing the trajectory of the state variables $\mathbf{x}(t)$. The multistep class of numerical methods employed in **deSolve** are based on Taylor expansions of $(V(t+\delta), R(t+\delta))$ at t for some small step δ. In **CollocInfer**, we use an alternative set of methods, known as *collocation* methods, which are based on approximating $(V(t), R(t))$ by basis expansions:

$$V(t) = \sum_{k=1}^{K_v} \phi_{vk}(t) c_{vk} \quad \text{and} \quad R(t) = \sum_{k=1}^{K_r} \phi_{rk}(t) c_{rk} \tag{4}$$

We write (4) in matrix notation as

$$V(t) = \mathbf{c}_v' \Phi_v(t) \quad \text{and} \quad R(t) = \mathbf{c}_r' \Phi_r(t) \tag{5}$$

by making the vector $\Phi_v(t) = (\phi_{v1}(t), \ldots, \phi_{vK_v}(t))^\top$, $\Phi_r(t) = (\phi_{r1}(t), \ldots, \phi_{rK_r}(t))^\top$ and collecting all the coefficients c_{vk} (c_{rk}) into the column vector $\mathbf{c}_v$ ($\mathbf{c}_r$). In this and many other situations we can use the same basis system for all variables, and if so we can simplify notation by designating a $K_r = K_V = K$ by 2 coefficient matrix $\mathbf{C}$ that has $\mathbf{c}_v$ and $\mathbf{c}_r$ in its first and second columns, respectively.

We now represent the trajectory in terms of this basis system. To do this, *collocation* methods seek to match the basis expansion with the differential equation at a pre-specified set of points. That is, they require

$$\Phi(0)\mathbf{C} = (V_0, R_0), \;\; D\Phi(t_q)\mathbf{C} = \mathbf{f}(\Phi(t_q)\mathbf{C}; t, \theta), \;\; q = 1, \ldots K-1,$$

where the *collocation points* t_q are generally spaced so as to cover the support of the basis functions. K collocation points are employed (including 0) so that the equations above can be satisfied exactly. We use this idea only indirectly and will often employ a few more collocation points. The interested reader is directed to Deuflhard and Bornemann (2000) for a more detailed presentation.

Although **CollocInfer** allows any choice of basis functions $\Phi(t)$, here we use the B-spline basis functions available through the **fda** library. B-splines are a set of functions that are polynomial between break points and are constrained to be smooth over the break points. We first define the range of the basis and the breaks. For the purpose of visualizing the basis expansion, here we use one break at every four time units, but we will use eight times as many when analyzing the data:

```
R> FhNrange <- c(0, 20)
R> breaks <- seq(0, 20, 4)
```

Then we specify the *order* of the polynomial segments, which is one more than the highest power in the polynomial. Here we use the default order four splines, which correspond to cubic polynomial segments. We now call a function in the **fda** package to assemble the basis

```
R> ExampleBasis <- create.bspline.basis(FhNrange, norder = 4,
+    breaks = breaks)
```

The basis can be visualized by

```
R> plot(ExampleBasis, xlab = "time", ylab = "Phi(t)")
```

resulting in the plot in Figure 2.

The resulting object `ExampleBasis` holds the specifications of the basis system. We can use the function `eval.basis` to obtain values of $\Phi(t)$ and its derivatives at specific t-values. **CollocInfer** only requires the values of this basis at particular time points, but it also works directly with a basis object such as `ExampleBasis`; we use this functionality in our initial examples.

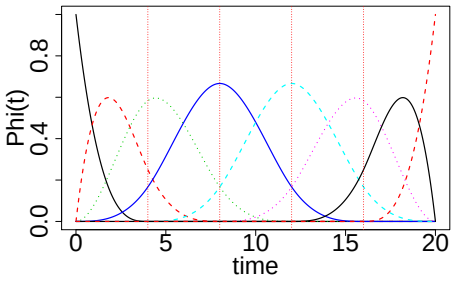

Figure 2: An example system of order 4 B-spline bases with breaks at every fourth time point. These functions are supported on a span of 5 knots and are made up of piecewise cubic polynomials between breaks with continuous second derivatives at breaks. Note that for realistic analyses, many more breaks than used here will be employed.

1.4. Basis expansions for nonparametric smoothing

As a final piece of background material, we briefly review nonparametric smoothing, in particular as applied in the **fda** package, extensively described in Ramsay *et al.* (2009). These also employ methods based on basis expansions and the commonalities between these methods and the collocation methods represent much of the inspiration for the methods in **CollocInfer**.

Smoothing data provides us a starting estimate of the matrix of coefficients $\mathbf{C}$. A classical means of smoothing is to estimate $\mathbf{c}_v$ by minimizing a penalized least squares criterion

$$\sum_{i=1}^{n} [y_{vi} - \Phi(t_i)\mathbf{c}_v]^2 + \lambda \int \left[D^2\Phi(t)\mathbf{c}_v\right]^2 dt, \tag{6}$$

and $\mathbf{c}_r$ can be similarly estimated. The first term in (6) measures agreement with data, just as in linear regressions, while the second imposes smoothness in the sense of having low curvature ($D^2x(t)$: the second derivative of $x(t)$). This second term is needed because $\Phi(t)$ is very flexible and can fit almost any data very closely, even when the data have a lot of noise. The multiplier λ is a smoothing parameter that controls the trade-off between data fitting and smoothness. There are many ways of selecting λ automatically, but for the moment we recommend trying a few values and adjusting it by eye. A more detailed treatment of these ideas can be found in Ramsay and Silverman (2005) and Ramsay, Hooker, and Graves (2009).

Let the 41 by 2 R matrix object `FhNdata` contain the observations of the two state variables at the 41 observation points. We now define a richer basis class using breaks at every observation point:

```
R> breaks <- seq(0, 20, 0.5)
R> FhNbasis <- create.bspline.basis(range = FhNrange, norder = 4,
+    breaks = breaks)
```

We can obtain initial estimates of $\mathbf{C}$ by employing the `smooth.fd` function in **fda**. The following command sets up an `fdPar` object `FhNfdPar` which specifies that the second derivative will be penalized and that the smoothing parameter λ in (6) has value 1:

```
R> FhNfdPar <- fdPar(FhNbasis, int2Lfd(2), 1)
```

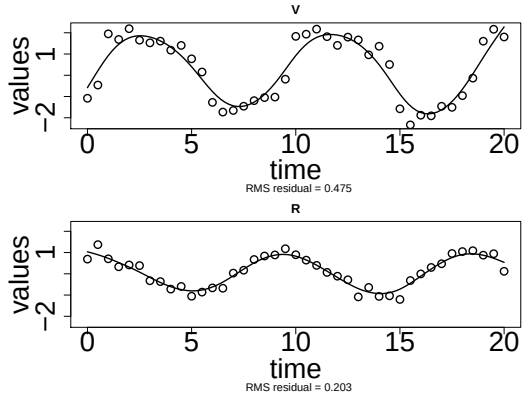

Figure 3: The result of `plotfit.fd` after smoothing data from the FitzHugh-Nagumo system.

Now we are ready to smooth the data:

```
R> DEfd0 <- smooth.basis(FhNtimes, FhNdata, FhNfdPar)$fd
```

We call the resulting function $\hat{\mathbf{x}}(t)$ returned in the functional data object `DEfd0`, and we examine the fit to the data using the `plotfit.fd` function in the **fda** package, the result of which is given in Figure 3:

```
R> plotfit.fd(FhNdata, FhNtimes, DEfd0)
```

These, of course, are not solutions to (3), but rather just smooths of the data. The coefficients of this smooth, which we will use later can be extracted from

```
R> coefs0 <- DEfd0$coef
```

2. Parameter estimation methods and basis expansions

In this section we introduce the methods implemented in **CollocInfer** and in the next section we will demonstrate their usage in R. We begin with a method called "gradient matching", a useful means of finding initial parameter estimates.

First we briefly comment on an obvious approach which searches for the set of parameters values such that the output of `lsoda` matches the observed data as well as possible. This approach might also involve searching over initial conditions $\mathbf{x}_0$. Such an approach, often referred to as "nonlinear least squares" or "NLS", can be problematical for three reasons:

1. The computational cost of repeatedly approximating the solution of the differential equation within an optimization routine can be considerable.

2. Frequently, the optimization of θ involves a criterion with many local minima and it is easy to get stuck, which necessitates stochastic optimization methods that will increase the computational cost above.

3. Often, we know that the systems under study only approximately follow the dynamic model, hence we may not want to insist on x_j being an exact solution to the ith differential equation.

Methods based on basis expansions and smoothing, which we will present later, avoid many of these problems.

2.1. Gradient matching with estimated derivatives

The idea behind what we term *gradient matching* is quite simple: suppose that we have produced a smooth of the data following (6). It is natural to find parameters θ so that the estimated $D\hat{\mathbf{x}}$ matches the right hand side of (1), $\mathbf{f}(\hat{\mathbf{x}}; t, \theta)$, as well as possible. Thus it makes sense to choose θ that minimizes the integrated sum of squared errors:

$$\text{ISSE}(\theta; \hat{\mathbf{x}}) = \sum_{j=1}^{d} \int \left[D\hat{x}_j(t) - f_j(\hat{\mathbf{x}}(t); t, \theta)\right]^2 dt \tag{7}$$

This approach, or variants on it, has been studied by numerous authors (Brunel 2008; Ellner, Seifu, and Smith 2002; Gugushvili and Klaassen 2012; Wu, Xue, and Kumar 2012) in recent statistical literature, but the idea goes back to Varah (1982) and Bellman and Roth (1971) as well as several other early re-inventions, and we would be frankly unsurprised to discover that Newton also thought of it.

In practice, we cannot evaluate the integral in ISSE directly, so instead we will rely on an approximation of the form

$$\widehat{\text{ISSE}}(\theta, \hat{\mathbf{x}}) = \sum_{q=1}^{Q} \sum_{j=1}^{d} w_q \left[D\hat{x}_j(t_q) - f_j(\hat{\mathbf{x}}(t_q); t_q, \theta)\right]^2$$

where t_q, $q = 1, \ldots, Q$ are *quadrature points* and w_q are corresponding *quadrature weights*. Simpson's rule is often employed here, but simply taking a large set of equally spaced t_q and giving them equal weight is usually also reasonable.

The above gradient matching approach has several advantages – in particular we do not need to repeatedly call `lsoda` to find values of θ. Frequently $\mathbf{f}(\mathbf{x}; t, \theta)$ is closer to linear than the solution to (1) meaning that ISSE is easier to work with. Finally, this framework allows $D\mathbf{x}$ to deviate somewhat from $\mathbf{f}(\mathbf{x}; t, \theta)$, hopefully without affecting the estimate of θ too much.

There are also some downsides. First, we must be able to estimate $\hat{x}_j$ for every state variable, which means that we must have enough high quality observations of this variable to put a smooth through it. This is often not the case in practice as some state variables may have no observations. For example in the above FitzHugh-Nagumo example, although we have simulated observations in both V and R above, only V will be available in real neural experiments. Another downside is that the form of (6) explicitly tries to reduce curvature. In Figure 3 we observe that the trajectories resulting from the FitzHugh-Nagumo equations may have quite sharp turns that will probably be smoothed over. Sharp curvatures are common in nonlinear ODE models, and bias in estimating the curves may induce bias in the parameter estimates.

Nonetheless, gradient matching is often a useful means of cheaply obtaining initial parameter estimates and these estimates can be used to start the generalized profiling procedure, which we describe below.

2.2. Generalized profiling: A data/equation fitting compromise

The generalized profiling methods in Ramsay *et al.* (2007) can alleviate the problems associated with gradient matching. First we observe that the penalty term in (6) tends to make the estimated $\hat{\mathbf{x}}$ look like a straight line satisfying $D^2\mathbf{x} = 0$. We might therefore think that, if we knew the value of θ, a better way to smooth would be via the criterion

$$\text{PENSSE}(\mathbf{x};\theta) = \sum_{i=1}^{n}\sum_{j\in C_O}(y_{ij} - x_j(t_i))^2 + \lambda \text{ISSE}(\theta,\mathbf{x}), \tag{8}$$

where the set C_O contains the indices j corresponding to observed states. Here the second penalty term ISSE defined in (7) tries to make $\mathbf{x}$ satisfy the differential equation (1) while the first term tries to make it look like the data. Since $\mathbf{x}(t) = \Phi(t)\mathbf{C}$, this is still really a problem of choosing coefficients $\mathbf{C}$. We might hope that, since we think that $\text{ISSE}(\theta,\mathbf{x})$ ought to be close to zero at the true trajectory, this will induce smaller bias. In particular, when $\mathbf{x}$ changes abruptly, (1) ought to model it and hence helps replicate this when we minimize PENSSE.

Note, importantly, that we do not require all of the x_j to have measurements to implement (8). A state variable is chosen to minimize $\text{ISSE}(\theta,\mathbf{x})$ only if it is measured. This way we make the trajectories of the measured state variables look as much like the ODE as possible.

So far, we have minimized $\text{PENSSE}(\mathbf{x};\theta)$ for $\mathbf{x}$, holding θ fixed. We now consider estimation of θ. We begin by pointing out an important distinction between the smoothing methods in (6) and $\text{PENSSE}(\mathbf{x},\theta)$. The penalty in (6) is intended to smooth out "wiggles" but otherwise let the curve follow the data. In contrast, we expect $\text{ISSE}(\theta,\mathbf{x})$ to have information about $\mathbf{x}$: if (1) holds, it should be zero at the correct θ. As a consequence, our estimated $\mathbf{x}$ should change considerably with different values of θ. Therefore, while in (6) we choose λ to be only large enough to stabilize our smooth, in $\text{PENSSE}(\mathbf{x},\theta)$ we expect λ to be large to enforce close agreement to the differential equation and reduce λ only to accommodate evident departures from it.

To proceed with smoothing, we denote by $\hat{\mathbf{x}}(t,\theta)$ the trajectory that minimizes $\text{PENSSE}(\mathbf{x},\theta)$ for each value of the parameter θ. We think of $\hat{\mathbf{x}}(t,\theta)$ as being like a solution of the ODE at θ. We can then choose θ to minimize squared error:

$$\text{SSE}(\theta) = \sum_{i=1}^{n}\sum_{j\in C_O}\left[y_{ij} - \hat{x}_j(t_i,\theta)\right]^2. \tag{9}$$

That is, we think of this method as nearly solving the ODE to obtain $\hat{\mathbf{x}}(t,\theta)$ and then fitting the "solution" to the data. This allows us to gain the advantages of gradient matching without incurring the same level of bias or requiring that all the components of $\mathbf{x}(t)$ have high quality observations. The methods here have been used and extended in Cao, Ramsay, and Fussmann (2008); Hooker (2009); Hooker, Ellner, de Vargas Roditi, and Earn (2011); Campbell, Hooker, and McAuley (2012); Qi and Zhao (2010) provides a theoretical analysis. Cao and Ramsay (2009) employs similar ideas in a mixed model framework.

In the next section we describe how to get the generalized profiling method running in **CollocInfer**.

3. An introduction to using the CollocInfer package

In order to obtain parameter estimates from **CollocInfer**, we set up some preliminary information. In particular, we define

- A basis system Φ.
- A function to evaluate the right hand side $\mathbf{f}$ of (2).
- Initial values to start off the iterative estimation of θ and $\mathbf{C}$.

We described how to obtain a B-spline basis system in Section 1.3. We will walk through the rest of the set-up process in the next two subsections.

3.1. Setting up the FitzHugh-Nagumo dynamic model

We first set up the function to evaluate $\mathbf{f}(\mathbf{x}; t, \theta)$. The package requires that $\mathbf{f}$ is a function with arguments:

`t:` a vector of evaluation times.

`x:` a matrix of evaluations of $\mathbf{x}(t)$ at the times in `t` with rows corresponding to time points.

`p:` a vector of parameters, θ.

`more:` a list of additional inputs that $\mathbf{f}$ might require.

The `more` argument allows the user to specify other inputs into $\mathbf{f}$. It can be unused, but should still be listed as an argument. Also the function should return a matrix of the same dimensions as `dx`.

For the FitzHugh-Nagumo equations, the function looks like

```
R> fhn.fun <- function(times, x, p, more) {
+    dx <- x
+    dx[, "V"] <- p["c"] * (x[, "V"] - x[, "V"]^3 / 3 + x[, "R"])
+    dx[, "R"] <- -(x[, "V"] - p["a"] + p["b"] * x[, "R"]) / p["c"]
+    return(dx)
+  }
```

The way we organize the evaluation function is somewhat different to the function `fhn.ode` that `lsoda` requires. Here we assume that the evaluation times in `times` are given by a vector and `x` is a matrix of values for the trajectory corresponding to the times `times`. The reason for this is that we examine all the values of $\mathbf{x}(t)$ at once, rather than sequentially, and employing the collection of evaluation points allows us to be more computationally efficient. We also only output the matrix `dx` of evaluations of `fhn.fun` rather than a list.

We assume that we have created a basis system `FhNbasis` as described in Section 1.3. We require the following input:

`data` an $n \times d$ matrix of data values. Missing values are listed as `NA`.

`times` a vector of n observation times.

Now we can set up some structures that **CollocInfer** needs. There is a simple function for the common choice of sum of squared errors criterion:

```
R> lambda <- 1000
R> profile.obj <- LS.setup(pars = FhNpars, fn = fhn.fun, lambda = lambda,
+    times = FhNtimes, coefs = coefs0, basisvals = FhNbasis)
```

We have chosen `lambda <- 1000` for convenience for the moment; we will discuss some ways to choose this later.

`LS.setup` returns a single list object with two named members, each of the list classes `lik` and `proc` that define the map from $\mathbf{x}$ to $\mathbf{y}$ and the map from $\mathbf{x}$ to $D\mathbf{x}$, respectively.

```
R> proc <- profile.obj$proc
R> lik <- profile.obj$lik
```

We ignore but will use implicitly some of the internal structures of these objects; see Section 5 for details. Note that by default **CollocInfer** employs finite differencing to estimate the derivatives needed in its optimization routines. However, finite differencing can be numerically unstable and the user can provide a named list of functions in place of `fhn.fun` above to evaluate also the first and second derivatives of `fhn.fun` (or whatever right hand side function is being used) with respect to `x` and `p`. See Section 8.1 for details. Given this framework, we can now start estimating parameters.

3.2. Obtaining initial parameter values via gradient matching

We showed how to obtain initial coefficients by minimizing (6) using the smoothing commands from the **fda** package. Next we use gradient matching to get an initial estimate of parameters. The function `ParsMatchOpt` minimizes $\text{ISSE}(\theta, \hat{\mathbf{x}})$ in (7) and returns a named list containing the optimized parameter values:

```
R> Pres0 <- ParsMatchOpt(FhNpars, coefs0, proc)
R> pars1 <- Pres0$pars
R> pars1
```

```
        a         b         c
0.2777251 0.1784665 1.4677403
```

The output list `ParsMatchOpt` also contains a member `Pres0$res`, the output of the minimization routine for finding the parameter values. By default `ParsMatchOpt` employs the R optimization function `nlminb`; several other options are also available in **CollocInfer**. Note here that while a and b are fairly close to their true value of 0.2, c is estimated at less than $1/2$ its true value. This is because c controls the relative speed of V and R and smoothing tends to dampen the V dynamics. The profiling methods employed below improve the precision of all of these estimates.

3.3. Running generalized profiling to obtain parameter estimates

A first step in the generalized profiling routines is to find **x** that minimizes the inner criterion PENSSE($\mathbf{x}; \theta$) in (8). This is achieved through the function `inneropt` and is called as follows:

```
R> Ires1 <- inneropt(FhNdata, times = FhNtimes, pars1, coefs0, lik, proc)
R> coefs1 <- Ires1$coefs
```

Strictly speaking, this is not particularly necessary, but it can be a useful check on both how long the overall procedure is likely to take and how well the model fits (see Section 3.6 below). The output of `inneropt` is a matrix of coefficients in `Ires1$coefs` and the result of the optimization routine in `Ires1$res`. By default it employs `nlminb` but a number of other routines can be set by `out.meth`.

We now carry out the profiling itself by

```
R> Ores2 <- outeropt(FhNdata, FhNtimes, pars1, coefs1, lik, proc)

  0: 17.905357: 0.277725 0.178467 1.46774
 10: 2.9261189: 0.193644 0.192393 2.93681
```

Again the output of `outeropt` is a list with members `pars`, `coefs` and `res` giving the optimum parameters, coefficients and whatever other information the optimization routine produces. Here again, `nlminb` is used as a default.

The estimated parameter vector is:

```
R> Ores2$pars

        a         b         c
0.1936466 0.1923805 2.9368073
```

While we have first produced `lik` and `proc` objects and then used them in `ParsMatchOpt`, `inneropt` and `outeropt`, we can directly call

```
R> Ires1.2 <- Smooth.LS(fhn.fun, FhNdata, FhNtimes, FhNpars, coefs0,
+    FhNbasis, lambda)
```

for the inner optimization and

```
R> Ores2.2 <- Profile.LS(fhn.fun, FhNdata, FhNtimes, FhNpars, coefs0,
+  FhNbasis, lambda)
```

to perform profiling directly. These functions both call `LS.setup` and return `lik` and `proc` objects. Because these two objects have enough utility in other functions, we recommend producing them independently.

3.4. Using `FPE` to choose the smoothing parameter λ

How to choose the smoothing λ in (8) is an important problem. In fact, **CollocInfer** provides the possibility of having a different λ for each state variable by setting λ to be a vector. One

approach is to examine visual diagnostics for when the fit to the data starts to degrade – Figure 4 provides an example of the fits obtained for different values of λ. An alternative is to keep increasing λ until the parameter estimates no longer change much (remember that the larger λ is, the closer $\hat{\mathbf{x}}(t, \theta)$ is to being solution of (1)). The second of these can start to produce bias when the basis expansion cannot approximate the solutions of the differential equation adequately. This source of bias is readily checked by applying profiling to a numerical solution of (1). Neither of these strategies are based on formal rules.

Ellner (2007) suggested the following idea: choose λ that predicts best using the differential equation (1). We can unpack this "best" with a recipe. For each value of λ

1. Conduct profiling to obtain $\hat{\theta}$ and $\hat{\mathbf{x}}(t, \hat{\theta})$.
2. For a set of starting times s_ℓ and ending times e_ℓ, $\ell = 1, \ldots, L$, obtain solutions to (1) with parameters $\hat{\theta}$ on the time interval $[s_\ell,\ e_\ell]$ with initial conditions $\hat{\mathbf{x}}(s_\ell, \hat{\theta})$.
3. Measure the squared error between these solutions and the observations in the interval $[s_\ell,\ e_\ell]$.

This was labelled *forwards prediction error* (FPE) because it describes how well exact solutions of (1) can be used for prediction over short-ish time intervals.

In **CollocInfer**, this recipe is implemented in the function `FPE`. For this function, besides the result of profiling, we need to define starting and ending times. These are given in a matrix where it is the *index* of the given observation times. In the case of the FitzHugh-Nagumo example, there are 41 observations and we will start from each time point and predict 10 ahead. This means that we define the object

```
R> whichtimes <- cbind(1:31, 11:41)
```

`whichtimes` is a matrix where the first column contains the indices of the observation times to start at, and the second column has the indices of the observation times to end at.

We now call

```
R> FPE <- forward.prediction.error(FhNtimes, FhNdata, Ores2$coefs,
+    lik, proc, Ores2$pars, whichtimes)
```

Generally, we would want to look at this for a selection of λ values to examine fit by eye. The code below cycles through values of λ increasing as powers of 10, reports the resulting parameter estimates and overlays the estimated trajectory on a plot of the data. In order to speed up computation we update the starting value for the parameters and coefficients to be the estimate with the previous value of λ. The code also waits for you to confirm before going on to the next λ (the function `fd` creates a functional data object that combines basis functions with their coefficients and allows easy plotting). The resulting plot is given in Figure 4. For clarity, we have only reproduced trajectories for three values of λ.

```
R> matplot(FhNtimes, FhNdata, pch = FhNvarnames)
R> lambdas <- 10^(0:7)
R> FPEs <- 0 * lambdas
R> temp.pars <- FhNpars
```

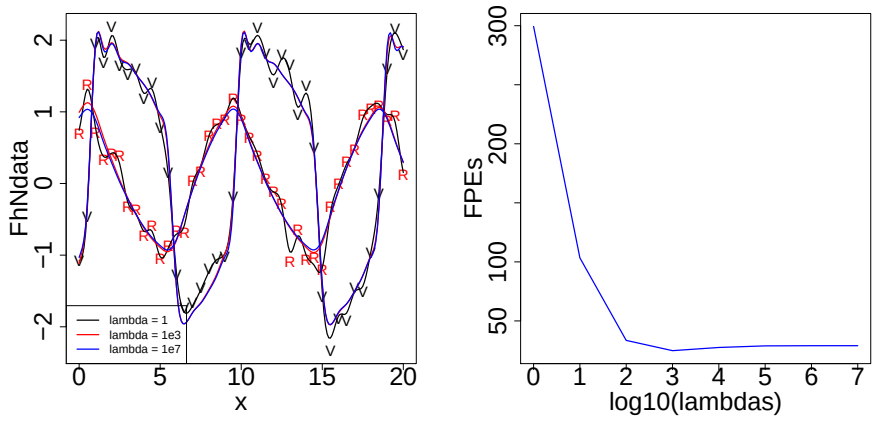

Figure 4: Left: estimated trajectories after profiling the FitzHugh-Nagumo data for three different values of λ. At $\lambda = 1$ small oscillations are visible that are smoothed out as λ increases and more weight is placed on matching the ODE. Right: forward prediction error over log lambda.

```
R> temp.coefs <- coefs0
R> for (ilam in 1:length(lambdas)) {
+    print(paste("lambda = ", lambdas[ilam]))
+    t.Ores <- Profile.LS(fhn.fun, FhNdata, FhNtimes, temp.pars,
+      temp.coefs, FhNbasis, lambdas[ilam])
+    print(t.Ores$pars)
+    temp.pars <- t.Ores$pars
+    temp.coefs <- t.Ores$coefs
+    t.fd <- fd(t.Ores$coefs, FhNbasis)
+    lines(t.fd, lwd = 2, col = ceiling(ilam / 2), lty = 1)
+    FPEs[ilam] <- forward.prediction.error(FhNtimes,
+      FhNdata, t.Ores$coefs, lik, proc, t.Ores$pars, whichtimes)
+    readline("Press Enter for the next lambda value")
+  }
```

The resulting values given below

```
R> FPEs

[1] 299.40290 103.41332 33.50794 24.81194 27.51236 28.86382 29.00435
[8] 29.01821
```

suggests our choice of $\lambda = 10^3$ is about appropriate. These values are plotted against the log of λ in Figure 4.

3.5. Using `IntegrateForward` to approximate an ODE Solution

We have used `lsoda` to solve differential equations to generate data, but it is somewhat frustrating to have to separately define different functions for the differential equation depending

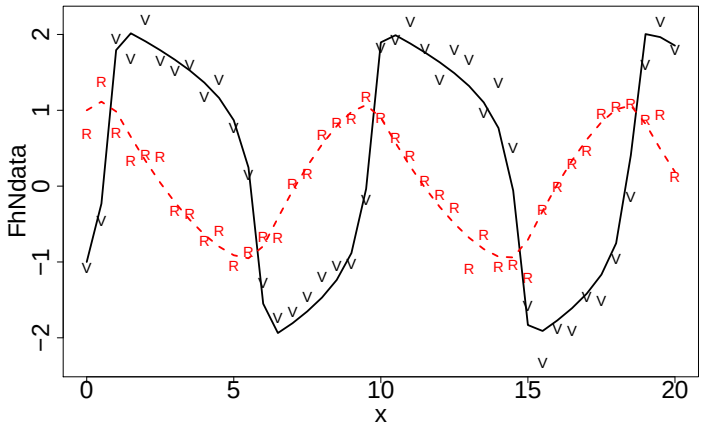

Figure 5: A comparison of the data to a solution of the FitzHugh-Nagumo equations with estimated parameters.

on what solver you use. To overcome this, **CollocInfer** includes a function `IntegrateForward` which uses functions in its format and passes them to `lsoda`.

We employ this function as an alternative to using `lsoda`. It returns a list with members `times` and `states` giving the times and values of the state variables respectively. We use this to compare the data to a solution of the ODE at the estimated parameters in Figure 5. (Here we will assume the initial conditions are known, although they could be extracted from the profiling solution.)

```
R> x0 <- c(-1, 1)
R> names(x0) <- FhNvarnames
R> sol1 <- IntegrateForward(x0, FhNtimes, Ores2$pars, proc)
R> matplot(FhNtimes, FhNdata, pch = c("V", "R"))
R> matplot(sol1$times, sol1$states, type = "l", add = TRUE)
```

It should be noted that a reason we did not employ `IntegrateForward` originally is that it requires setting up a `proc` object which makes less sense pre-data. Of course, if there is experimental data, one will not have needed to solve an ODE and can start from `LS.setup` and use this utility directly.

3.6. Using `CollocInferPlots` for diagnostic plots

We now assess how well the profiling procedures are doing. There are two things that we could look at :

- How well the estimated trajectory $\hat{\mathbf{x}}(t, \theta)$ fits the data.
- How well $D\hat{\mathbf{x}}(t, \theta)$ matches $\mathbf{f}(\hat{\mathbf{x}}; \theta)$.

The function `CollocInferPlots` produces basic diagnostic plots for these and can be called as follows

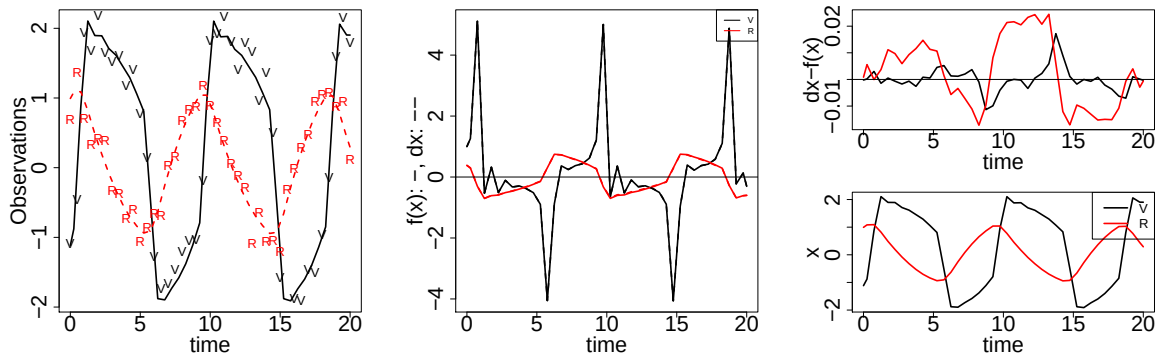

Figure 6: Diagnostic plots for the fit of the FitzHugh-Nagumo System. Left: The data plotted along with estimated trajectories. Middle: The estimated derivative of the trajectory and the derivative calculated by the ODE. Right top: The difference between the estimated and ODE derivatives. Right bottom: The estimated trajectories for comparison with the plot above.

```
R> out1 <- CollocInferPlots(Ores2$coefs, Ores2$pars, lik, proc,
+     times = FhNtimes, data = FhNdata)
```

This produces the plots in Figure 6. The first plot displays the data and the estimated trajectory which enables us to observe if (1) pulls us too far from the data. The second plot gives the $D\hat{\mathbf{x}}(t,\theta)$ as dashed lines and the value of $\mathbf{f}(\hat{\mathbf{x}},\theta)$ at that trajectory value as solid lines. In this case they are nearly on top of each other, but when they are further apart this provides a good indication of what parts of the trajectory do not fit. This is made more explicit in the third plot where the upper panel gives the difference between $D\hat{\mathbf{x}}(t,\theta)$ and $\mathbf{f}(\hat{\mathbf{x}};\theta)$ and the lower is $\hat{\mathbf{x}}(t,\theta)$, which enables a rough comparison of under what conditions the mismatch between derivatives appears to be large. In this case there are hints that the mismatch in the derivative for R and its equation may be associated with the current value of V, but the overall level of mismatch is very small.

3.7. Using `Profile.covariance` to estimate the sample variance

Finally, it is useful to evaluate the statistical uncertainty in our parameter estimates. This is obtained through an Newey-West estimate (Newey and West (1987)) of the covariance of the score function for the outer optimization, the details of which are somewhat more technical and are given in Hooker *et al.* (2011).

The following function call will perform the estimate

```
R> covar <- Profile.covariance(Ores2$pars, times = FhNtimes, data = FhNdata,
+     coefs = Ores2$coefs, lik = lik, proc = proc)
```

and return

```
R> covar

              [,1]          [,2]          [,3]
[1,] 0.0006038836 0.0025175389 0.0001234712
```

```
[2,] 0.0025175389 0.0140052441 0.0006037352
[3,] 0.0001234712 0.0006037352 0.0007241856
```

In order to construct confidence intervals we add and subtract two standard deviations

```
R> CIs <- cbind( Ores2$pars - 2 * sqrt(diag(covar)),
+    Ores2$pars + 2 * sqrt(diag(covar)) )
R> rownames(CIs) <- FhNparnames
R> CIs

         [,1]      [,2]
a  0.14449851 0.2427947
b -0.04430699 0.4290680
c  2.88298594 2.9906287
```

3.8. Using `FitMatchOpt` when only some state variables are observed

So far we have used an example where both V and R have observations. However, for many systems some components of the model cannot be measured; indeed in neural experiments observations are generally given only for V. Our methods work in this situation as well. However, more work needs to be done to obtain initial coefficient and parameter estimates.

We set up some data with only V by simply converting the data for R to take value `NA`

```
R> data2 <- FhNdata
R> data2[, 2] <- NA
```

and we go back to the original nonparametric smooth from which we obtained `coefs0` and set all the coefficients for R to be zero

```
R> coefs0.2 <- coefs0
R> coefs0.2[, 2] <- 0
```

The first thing is to compute some decent starting values for `coefs0.2[, 2]`. To do this, we assume that we have initial parameter values. Here we will use `pars1`, the results of gradient matching. In doing this we want to remind the reader that gradient matching cannot be performed without measurements of R – this is just a convenient set of values to use.

Next we try to estimate the second column of `coefs0.2` so that the whole trajectory matches (1) as well as possible with the first column of coefficients held fixed. That is, we optimize $\text{ISSE}(\theta, \mathbf{x})$ but over some of the components of $\mathbf{x}$ rather than over θ. We do this with the following function

```
R> Fres3 <- FitMatchOpt(coefs0.2, 2, pars1, proc)
```

The second argument here specifies which columns of `coefs0.2` to optimize. Note that since the components that have been smoothed are held fixed, the `lik` object is not needed. The output of `FitMatchOpt` is a complete set of components in `Fres3$coefs` as well as `Fres3$res`

which gives the output of the optimization routine. (Again `nlminb` is used by default, but other options are available.)

Following this, we proceed with profiling as before. The reader can verify that the following commands produce sensible point and interval estimates for the three parameters

```
R> Ores4 <- outeropt(FhNdata, FhNtimes, pars1, Fres3$coefs, lik, proc)
R> Ores4$pars

        a         b         c
0.1936414 0.1923929 2.9367916

R> out2 <- CollocInferPlots(Ores4$coefs, Ores4$pars, lik, proc,
+    times = FhNtimes, data = data2)
R> covar4 <- Profile.covariance(Ores4$pars, times = FhNtimes, data = data2,
+    coefs = Ores4$coefs, lik = lik, proc = proc)
R> CI4 <- cbind(Ores4$pars - 2 * sqrt(diag(covar4)),
+    Ores4$pars + 2 * sqrt(diag(covar4)) )
R> rownames(CI4) <- FhNparnames
R> CI4

        [,1]      [,2]
a  0.1070975 0.2801853
b -0.2272742 0.6120600
c  2.4305522 3.4430311
```

3.9. Real world experience: A chemostat experiment

Our methods, like most, work very well when tested on data simulated from the model the method is designed for. Working with real-world data can be a very different experience. In this section we deal with data generated from a laboratory-based ecological experiment. In this system, an algae of genus *Chlorella*, C, is grown in a *chemostat*, a large glass test-tube to which a nutrient-rich medium is continuously added, and from which the contents are removed (including the algae) at a constant rate. The growth of the algal population is limited by nutrition in the ecology and by predation by rotifers, *Brachionus*, B – a genus of microscopic animals. The rotifers reproduce according to how much algae they consume and die either from natural causes or when they are removed from the tank. Data from a run of this experiment are taken from Becks, Ellner, Jones, and Hairston (2010) and plotted in Figure 7.

We choose to model these in terms of $C^* = \log(C)$ and $B^* = \log(B)$ so that DC represents the *per algae* rate of growth in the population (see Section 4 for doing this automatically). We employ the Rosenzweig-MacArthur model (Rosenzweig and MacArthur 1969) which describes the growth of algae in terms of reproduction in which we think of each algae reproducing at rate $\rho(1 - C/\kappa_C)$ (note that we use C rather than C^*) so that as the population reaches its carrying capacity κ_C, the growth rate slows down due to resource limitations, and predation by rotifers which we parameterize by $\gamma B/(\kappa_B + C)$ where the predation rate increases as there are more rotifers, but the denominator ensures that the rate of predation per algae decreases

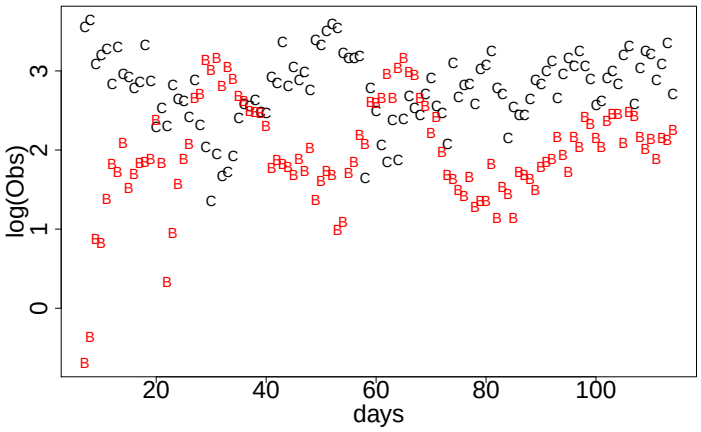

Figure 7: Data on log algal numbers (C) and log rotifer numbers (B) collected from an experiment in laboratory-based ecology.

as the number of algae increases reaching the limit of the rotifer's capacity to consume. We model per-population rotifer dynamics in terms of increase following algal consumption by $\chi\gamma C/(\kappa_B+C)$ and a death rate δ. This gives us the following differential equations (expressed in the logarithmic terms of C^* and B^*):

$$DC^* = \rho\left(1 - e^{C^*}/\kappa_C\right) - \frac{\gamma e^{B^*}}{\kappa_B + e^{C^*}}$$

$$DB^* = \frac{\chi\gamma e^{C^*}}{\kappa_B + e^{C^*}} - \delta.$$

We instantiate this in R in the following model:

```
R> RosMac <- function(t, x, p, more) {
+    p <- exp(p); x <- exp(x)
+    dx <- x
+    dx[, "C"] <- p["rho"] * (1 - x[, "C"] / p["kappaC"]) -
+      p["gamma"] * x[, "B"] / (p["kappaB"] + x[, "C"])
+    dx[, "B"] <- p["chi"] * p["gamma"] * x[, "C"] /
+      (p["kappaB"] + x[, "C"]) - p["delta"]
+    return(dx)
+  }
```

Note that we have exponentiated both `x` (since the dynamics are all in terms of the non-logged quantities) and the parameter vector `p`. In fact we know that `p` is positive and the parameters have very different magnitudes, estimating their log quantities then both ensures that they take sensible values and helps to make the process more numerically stable.

The data can be obtained from

```
R> data("ChemoRMData")
```

which we convert to

```
R> time <- ChemoRMTime
R> data <- log(ChemoRMData)
```

and we set up the framework to estimate the model as follows. We define initial parameters

```
R> varnames <- RMvarnames
R> parnames <- RMparnames
```

basis functions

```
R> rr <- range(time)
R> breaks <- seq(rr[1], rr[2], by = 1)

R> mids <- c(min(breaks), breaks[1:(length(breaks) - 1)] + 0.5, max(breaks))

R> ChemoRMbasis <- create.bspline.basis(rr, norder = 4, breaks = breaks)
```

and an initial set of parameters and coefficients

```
R> fd0 <- smooth.basis(time, data, fdPar(ChemoRMbasis, int2Lfd(2), 10))
R> coef0 <- fd0$fd$coef
R> colnames(coef0) <- varnames
R> pars <- log(ChemoRMPars)
R> names(pars) <- parnames
```

Examining the `ChemoRMPars` vector, while we have employed 1 as an initial estimate for γ, κ_B, χ and δ (0 after taking logs) we can (and have) obtained estimates of ρ and κ_C from experiments without rotifer predation and we will fix these below. To do so, we define `activepars` to be the indices of the parameters in `p` that we wish to estimate.

```
R> activepars <- 3:6
```

this can be called in the `active` argument of `ParsMatchOpt` and `outeropt` as well as `Profile.LS`. From here we set up the profiling objects

```
R> out <- LS.setup(pars = pars, coefs = coef0, basisvals = ChemoRmbasis,
+    fn = RosMac, lambda = c(5e4, 5e2), times = time)
R> lik <- out$lik
R> proc <- out$proc
```

and start by obtaining an initial estimate of parameters. Here the `active` elements tell `ParsMatchOpt` which elements of `pars` it should estimate:

```
R> res1 <- ParsMatchOpt(pars, coef0, proc, active = activepars)
```

Then we call the profiling procedure

```
R> res2 <- outeropt(data, time, res1$pars, coef0, lik, proc,
+    active = activepars)
```

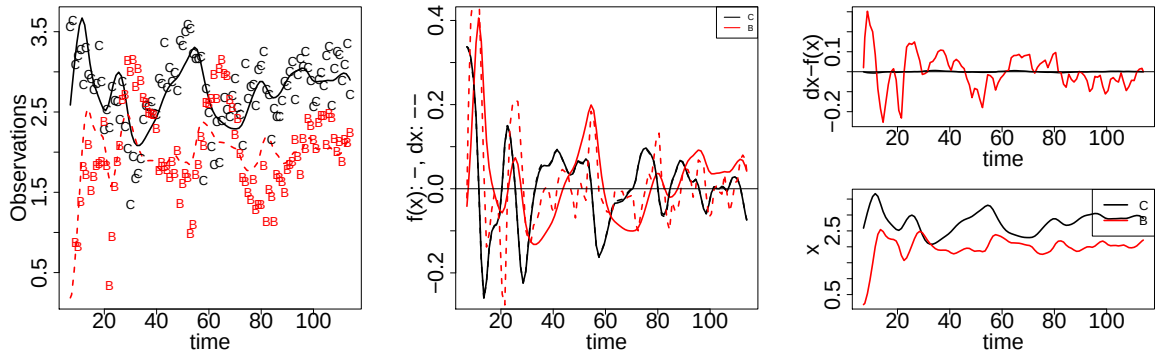

Figure 8: Diagnostic plots for the fit of the Rosenzweig-MacArthur System to the chemostat data. See Figure 6 for explanation.

and construct diagnostic plots

```
R> out2 <- CollocInferPlots(res2$coefs, res3$pars, lik, proc,
+    times = time, data = data)
```

which are reproduced in Figure 8.

We also provide confidence intervals:

```
R> covar <- Profile.covariance(res2$pars, times = time, data = data,
+    coefs = res2$coefs, lik = lik, proc = proc, active = activepars)
R> CIs <- cbind(res2$pars[activepars] - 2 * sqrt(diag(covar)),
+    res2$pars[activepars] + 2 * sqrt(diag(covar)))
R> rownames(CIs) <- parnames[activepars]
```

In this case we work these out for the log parameters and then re-exponentiate:

```
R> exp(CIs)
```

```
               [,1]         [,2]
gamma  5.157680e+07 5.421585e+07
kappaB 9.318361e+08 1.114322e+09
chi    3.171171e-01 3.474296e-01
delta  2.452262e-01 2.984662e-01
```

and check that we employed the right trade-off between the model and the data. We use a 5-day look-ahead window and choose to multiply our original weighting by a factor ranging from 0.1 to 10:

```
R> whichtimes <- cbind(1:102, 6:107)
R> lambdafac <- c(0.1, 0.5, 1, 5, 10)
R> FPEs <- 0 * lambdafac
R> temp.pars <- res2$pars
R> temp.coefs <- res2$coefs
```

```
R> for(ilam in 1:length(lambdafac)) {
+    t.res <- Profile.LS(RosMac, data, time, temp.pars, temp.coefs,
+      ChemoRMbasis, lambdafac[ilam] * c(5e4, 5e2),
+      active = activepars, out.meth = "nlminb")
+    FPEs[ilam] <- forward.prediction.error(time, data, t.res$coefs,
+      lik, proc, t.res$pars, whichtimes)
+    temp.pars <- t.res$pars
+    temp.coefs <- t.res$coefs
+  }
```

which produces

```
R> FPEs

[1] 263.7641 252.7731 251.1961 251.8647 253.9571
```

The above outputs confirms our choice of weights.

4. Some further details and a more complex model

So far we have mainly dealt with systems of differential equations that are directly observed, although in practice we may be missing observations for some state variables. **CollocInfer** can deal with more complicated data designs and modeling situations, which can be useful in some settings. In particular, we will discuss three cases here:

1. We know that all the state variables are positive, and often it is useful to use an ODE representation for the log state variables $\mathbf{z} = \log \mathbf{x}$. This is especially important for problems where data values vary by orders of magnitude from one variable to another, as is often the case with models for spread of disease and population dynamics.

2. We do not observe the system directly, but rather we measure some transformation of it, or some aggregation of certain variables.

3. We have replicated observations of the system.

These situations arise frequently in practice, and can all be dealt with by using the basic setup utilities in **CollocInfer**.

To illustrate these cases, we use an extension of the Rosenzweig-MacArthur system described previously. In this system, we propose two types of algae C_1 and C_2, each of which has similar dynamics to the single algal species in the system above. However, they have different reproduction rates and suffer different predation risks.

The system is of interest because at some parameter values we can maintain two types of algae, even though they face the same pressure and one ought to out-compete the other. This is because we set the first algae to have a defense against predation, but make it pay a cost for that defense in terms of the efficiency of its reproduction. This means that when there are no rotifers around, the second algae has an advantage, but when there are rotifers, the first does and, in fact, the system produces oscillations that keep all three species present.

We model the systems in the following equations:

$$\begin{aligned} DC_1 &= \rho_1 C_1 \left(1 - \frac{C_1}{\kappa_{C_1}} - \frac{C_2}{\kappa_{C_2}}\right) - \frac{\pi\gamma C_1 B}{\kappa_B + \pi C_1 + C_2} \\ DC_2 &= \rho_2 C_2 \left(1 - \frac{C_1}{\kappa_{C_1}} - \frac{C_2}{\kappa_{C_2}}\right) - \frac{\gamma C_2 B}{\kappa_B + \pi C_1 + C_2} \\ DB &= \frac{\chi\gamma B\left(\pi C_1 + C_2\right)}{\kappa_B + \pi C_1 + C_2} - \delta B. \end{aligned} \tag{10}$$

Here ρ_1 and ρ_2 are differential rates of growth of the algae, and the nutrients (hence algal growth) are exhausted at $C_1/\kappa_{C_1} + C_2/\kappa_{C_2} = 1$. By setting $\kappa_{C_1} < \kappa_{C_2}$, we assume that the overall carrying capacity for C_1 is less than for C_2. Note that we no longer model log quantities (we will do this automatically below) so the right hand side equations are multiplied by C and B respectively. We will convert to using log rates again below, but will use **CollocInfer**'s internal conversion for this.

For predation, the rotifers have nutrients given by $\pi C_1 + C_2$ where π represents a fraction of the defended C_1 that are available for predation. Here the second term for C_2 gives a predation rate $\gamma B/(\kappa_B + \pi C - 1 + C_2)$ where κ_B is the amount of available algae at which the predation rate is halved. For the C_1 equation the relevant quantity of algae is πC_1.

In the final equation, the rotifers turn consumed algae into new rotifers with efficiency χ, but also die at rate δ. These equations are coded in a manner suitable for **CollocInfer** as follows:

```
R> RosMac2 <- function(t, x, p, more) {
+    p <- exp(p)
+    dx <- x
+    dx[, "C1"] <- p["rho1"] * x[, "C1"] *
+      (1 - x[, "C1"] / p["kappaC1"] - x[, "C2"] / p["kappaC2"]) -
+      p["pi"] * p["gamma"] * x[, "C1"] * x[, "B"] /
+      (p["kappaB"] + p["pi"] * x[, "C1"] + x[, "C2"])
+    dx[, "C2"] <- p["rho2"] * x[, "C2"] *
+      (1 - x[, "C1"] / p["kappaC1"] - x[, "C2"] / p["kappaC2"]) -
+      p["gamma"] * x[, "C2"] * x[, "B"] /
+      (p["kappaB"] + p["pi"] * x[, "C1"] + x[, "C2"])
+    dx[, "B"] <- p["chi"] * p["gamma"] *
+      (p["pi"] * x[, "C1"] + x[, "C2"]) * x[, "B"] /
+      (p["kappaB"] + p["pi"] * x[, "C1"] + x[, "C2"]) -
+      p["delta"] * x[, "B"]
+    return(dx)
+  }
```

Here we know that all the parameters should be positive, so we have hard-coded that we will estimate the log parameters. This also helps by reducing the difference in the scaling of the parameters so that their numerical values (after taking logs) are comparable. To generate data, we set these to be

```
R> RMpars <- c(0.2, 0.025, 0.125, 2.2e4, 1e5, 5e6, 1, 1e9, 0.3)
R> RMParnames <- c("pi", "rho1", "rho2", "kappaC1", "kappaC2",
+    "gamma", "chi", "kappaB", "delta")
```

```
R> logpars <- log(RMpars)
R> names(logpars) <- RMParnames
```

We start with some initial conditions

```
R> RMVarnames <- c("C1", "C2", "B")
R> x0 <- c(50, 50, 2)
R> names(x0) <- RMVarnames
```

There are a few things that we need to know about this system:

1. The state variables are all population sizes and hence are always positive. However the populations can become very small, which creates numerical instability near $x_j = 0$. Consequently, it will be useful to model an ODE for $\mathbf{z} = \log \mathbf{x}$. This also helps to reduce numerical instability caused by the very different scales of C and B.

2. As in many real-world systems, we cannot distinguish C_1 from C_2 (in fact, a problem of current research is to detect if C_2 is there at all). So we really only measure $C_1 + C_2$.

3. However, we can repeat the experiment several times so that we have a replicated time series.

Methods for accounting for each of these within **CollocInfer** are demonstrated in turn in the subsections below.

4.1. Dynamics on the log scale

The first observation is that all the state variables are strictly positive, so it may be useful to model $\mathbf{z} = \log \mathbf{x}$. By employing the chain rule, we write out a new ordinary differential log-scale equation:

$$D\mathbf{z} = \frac{\mathbf{f}\left(e^{\mathbf{z}}; \theta\right)}{e^{\mathbf{z}}} \tag{11}$$

where the ratio is taken elementwise. The Rosenzweig-MacArthur model (Rosenzweig and MacArthur 1969) is implemented below incorporating the log transform in a form suitable for use with `lsoda`:

```
R> RosMac2ODE <- function(t, z, p) {
+    p <- exp(p)
+    x <- exp(z)
+    dx <- x
+    dx["C1"] <- p["rho1"] * x["C1"] *
+      (1 - x["C1"] / p["kappaC1"] - x["C2"] / p["kappaC2"]) -
+      p["pi"] * p["gamma"] * x["C1"] * x["B"] /
+      (p["kappaB"] + p["pi"] * x["C1"] + x["C2"])
+    dx["C2"] <- p["rho2"] * x["C2"] *
+      (1 - x["C2"] / p["kappaC2"] - x["C1"] / p["kappaC1"]) -
+      p["gamma"] * x["C2"] * x["B"] /
+      (p["kappaB"] + p["pi"] * x["C1"] + x["C2"])
```

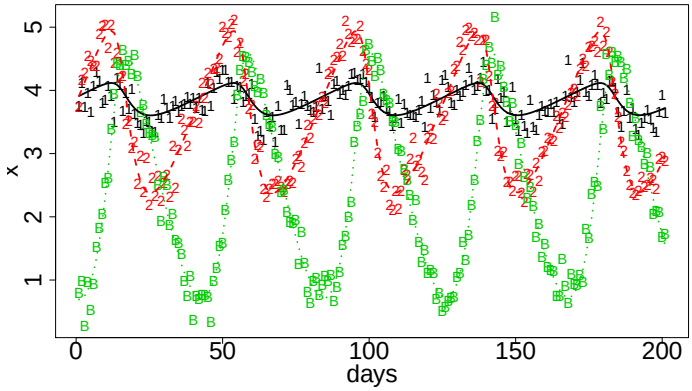

Figure 9: Data generated from the Rosenzweig-MacArthur Ordinary Differential Equation.

```
+     dx["B"] <- p["chi"] * p["gamma"] * (p["pi"] * x["C1"] + x["C2"]) *
+       x["B"] / (p["kappaB"] + p["pi"] * x["C1"] + x["C2"]) -
+       p["delta"] * x["B"]
+     return(list(dx / x))
+   }
```

With the above function we now generate data by solving the ordinary differential equation and adding noise. For the moment, we pretend that we can measure each of the three species in the chemostat independently and that the experiment lasts 200 days and is sampled daily. We have plotted the outcome of this in Figure 9.

```
R> time <- 0:200
R> res0 <- lsoda(log(x0), time, RosMac2ODE, p = logpars)
R> matplot( exp(res0[, 2:4]), type = "l")
R> data <- res0[, 2:4] + 0.2 * matrix(rnorm(603), 201, 3)
R> matplot(data)
R> matplot(res0[, 2:4], type = "l", add = TRUE)
```

Then we employ the same commands as before to set up basis expansions, smooth the data

```
R> rr <- range(time)
R> breaks <- seq(rr[1], rr[2], by = 1)
R> RMbasis <- create.bspline.basis(rr, breaks = breaks)
```

and obtain an initial set of parameters and coefficients

```
R> coef0 <- smooth.basis(time, data, fdPar(RMbasis, int2Lfd(2), 10))$fd$coef
R> colnames(coef0) <- RMVarnames
```

To set up this equation on the log scale, we could transform `RosMac2ODE` into a form appropriate for **CollocInfer**. However, this requires writing out a new function and instead, we employ the `posproc = TRUE` option in `LS.setup` which makes the transformation automatically. But

because our data is already measured on the log-scale since we added noise to the solutions of `RosMac2ODE`, we do not need the corresponding `poslik = TRUE` option. This is implemented in

```
R> out <- LS.setup(pars = logpars, coefs = coef0, basisvals = RMbasis,
+    fn = RosMac2, lambda = 1e5, times = time, posproc = TRUE)
R> lik <- out$lik
R> proc <- out$proc
```

Now we employ calls to functions `ParsMatchOpt` and `outeropt`

```
R> res1 <- ParsMatchOpt(logpars, coef0, proc)
R> res3 <- outeropt(data, time, res1$pars, coef0, lik, proc)
```

The following output from function `outeropt` has been rounded to two decimals to fit into this page:

```
  0: 25.27: -1.48 -3.50 -2.05 19.47 6.42 15.40 0.027 20.74 -1.22
 10: 24.44: -1.50 -3.49 -2.01 19.47 6.47 15.41 0.035 20.73 -1.19
 20: 24.10: -1.51 -3.55 -2.06 19.47 7.26 15.42 0.019 20.73 -1.20
 30: 24.03: -1.52 -3.59 -2.08 19.47 8.63 15.42 0.019 20.73 -1.20

R> exp(res3$pars)

pi       rho1     rho2     kappaC1  kappaC2
2.19e-01 2.77e-02 1.25e-01 2.84e+08 5.58e+03

gamma    chi      kappaB   delta
4.96e+06 1.02e+00 1.01e+09 3.01e-01
```

and we call `CollocInferPlots` to examine goodness of fit in Figure 10

```
R> out3 <- CollocInferPlots(res3$coefs, res3$pars, lik, proc,
+    times = time, data = data)
```

The same option can be used with `smooth.LS` and `profile.LS`.

4.2. Indirectly observed systems: the sum of C_1 and C_2

In chemostat experiments it is often impossible to distinguish C_1 from C_2, so we only get to observe their sum. We will mimic this by exponentiating the data we generated for these (remember it is currently on a log scale) taking the sum and then re-logging:

```
R> data2 <- cbind(log(exp(data[, "C1"]) + exp(data[, "C2"])), data[, "B"])
```

See Figure 11. Notice that these data are no longer directly comparable to the states of the ordinary differential equation. In this case we can write a function to go from states to observations by exactly repeating the process above. The structure of this function look exactly like the functions we employ for the right hand side of an ODE:

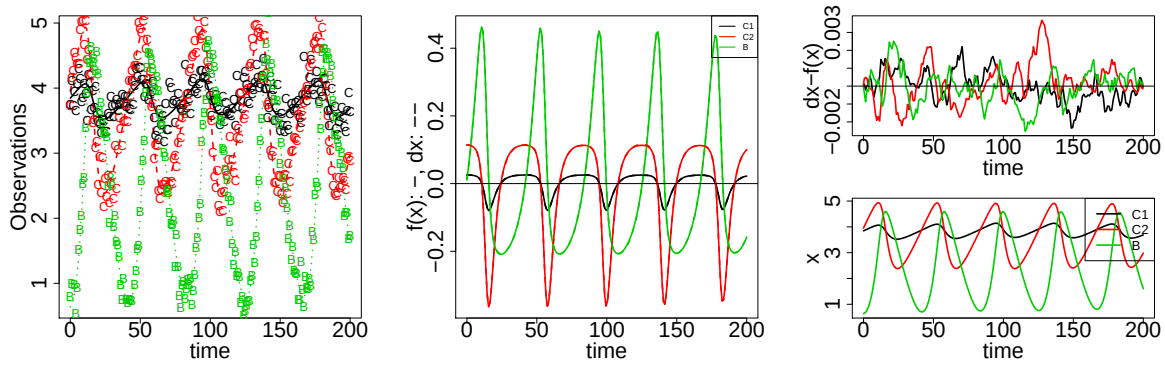

Figure 10: Diagnostic Plots for the Rosenzweig-MacArthur system.

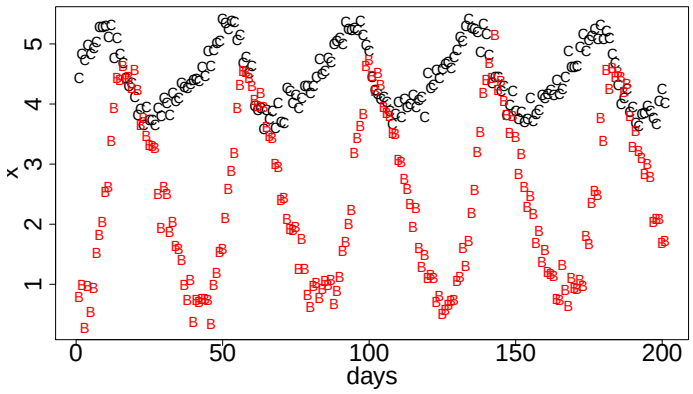

Figure 11: Logged Rosenzweig-MacArthur data when only B and $C_1 + C_2$ can be observed.

```
R> RMobsfn <- function(t, x, p, more) {
+     x <- exp(x)
+     y <- cbind( x[, "C1"] + x[, "C2"], x[, "B"])
+     return(log(y))
+  }
```

In this case, the observation model does not change with time `t`, parameters `p` and we have no additional information to add in `more`. We also add this function as the `likfn` argument to `LS.setup`, `smooth.LS` or `profile.LS`:

```
R> out <- LS.setup(pars = logpars, coefs = coef0, basisvals = RMbasis,
+     fn = RosMac2, lambda = 1e5, times = time, posproc = TRUE,
+     likfn = RMobsfn)
R> lik2 <- out$lik
R> proc2 <- out$proc
```

In this case it is unreasonable to assume that we can start from `coef0` as a pre-smooth since we do not have separate data from C_1 and C_2 so we set the corresponding coefficients to zero:

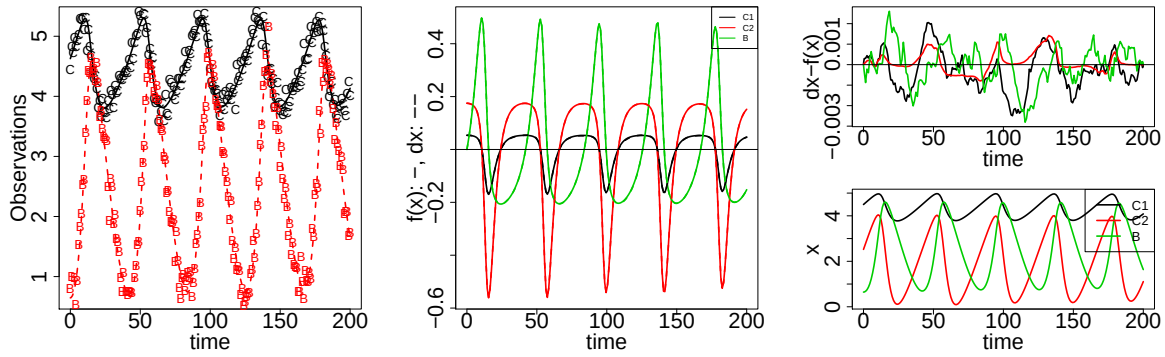

Figure 12: Diagnostic plots for indirectly observed Rosenzweig-MacArthur data.

```
R> coef02 <- coef0
R> coef02[, 1:2] <- 0
```

but we keep the coefficients for B. From this we start by using `FitMatchOpt` as above and then profiling

```
R> Fres3 <- FitMatchOpt(coef02, 1:2, res1$pars, proc2)
R> res32 <- outeropt(data2, time, res1$pars, Fres3$coefs, lik2, proc2)
R> exp(res32$pars)
```

```
pi        rho1      rho2      kappaC1  kappaC2
3.03e-01 5.84e-02 1.91e-01 2.84e+08 1.35e+03
gamma     chi        kappaB   delta
6.17e+06 1.092e+00 8.26e+08 3.24e-01
```

and examine the usual diagnostic plots; see Figure 12.

```
R> out32 <- CollocInferPlots(res32$coefs, res32$pars, lik2, proc2,
+    times = time, data = data2, datanames = c("C", "B"))
```

4.3. Replicated experiments

In some instances it is possible to repeat experimental runs and observe how different some results might be. These can be accommodated in **CollocInfer**, although most situations will require the user to work more carefully with the structure of `lik` and `proc` objects. See Section 5 for more details.

However, `LS.setup`, `smooth.LS` and `profile.LS` will work if each repetition of the experiment has the same observation times and the same time domain. This means that we can use the same basis expansion for all experimental repeats. We achieve this structure by using the collection of all observation times with `NA` entries where observation times do not match up, but this approach will result in more numerical effort than necessary if, for example, the experiments run for very different durations.

We mimic this by running a new chemostat experiment starting from different initial conditions. For the sake of simplicity, we continue to pretend that we measure all the state variables.

```
R> x03 <- c(15, 25, 4)
R> names(x03) <- RMVarnames
R> res03 <- lsoda(log(x03), time, RosMac2ODE, p = logpars)
R> data03 <- res03[, 2:4] + 0.2 * matrix(rnorm(603), 201, 3)
```

We now put this together with `data` in a three-dimensional array where the dimensions correspond to times, replicate and dimension in that order. The following commands give us two experiments of data.

```
R> alldat <- array(0, c(201, 2, 3))
R> alldat[, 1, ] <- data
R> alldat[, 2, ] <- data03
```

We also need an initial three dimensional coefficient array, which we could get by smoothing the new data and putting the resulting coefficients with `coef0`

```
R> coef3 <- smooth.basis(time, data03, fdPar(RMbasis, int2Lfd(2), 10))$fd$coef
R> coefs <- array(0, c(dim(coef3)[1], 2, 3))
R> coefs[, 1, ] <- coef0
R> coefs[, 2, ] <- coef3
```

and we now use these in the call to `LS.setup`

```
R> out <- LS.setup(pars = logpars, coefs = coefs, basisvals = RMbasis,
+    fn = RosMac2, lambda = 1e5, times = time, data = alldat,
+    posproc = TRUE, names = RMVarnames)
R> lik3 <- out$lik
R> proc3 <- out$proc
```

The output of `LS.setup` includes `coefs` and `data`. These are the same as what we put in but are re-formatted so that the second dimension is stacked. This allows **CollocInfer** to pretend that we have only one experiment, but twice as long and there might be a sudden jump in the middle (the same strategy is used if you want to set things up manually below). We use these outputs to call the functions `inneropt` and `outeropt`:

```
R> res13 <- inneropt(data = out$data, times = out$times, pars = res1$pars,
+    coefs = out$coefs, lik = lik3, proc = proc3)
R> res33 <- outeropt(data = out$data, times = out$times, pars = res1$pars,
+    coefs = res13$coefs, lik = lik3, proc = proc3)
R> exp(res33$pars)

pi      rho1    rho2     kappaC1  kappaC2
2.17-01 2.70-02 1.25e-01 2.84e+08 1.83e+04

gamma    chi      kappaB   delta
5.00e+06 9.81e-01 1.00e+08 2.99e-01

R> out3 <- CollocInferPlots(res33$coefs, res33$pars, lik3, proc3,
+    times = out$times, data = out$data)
```

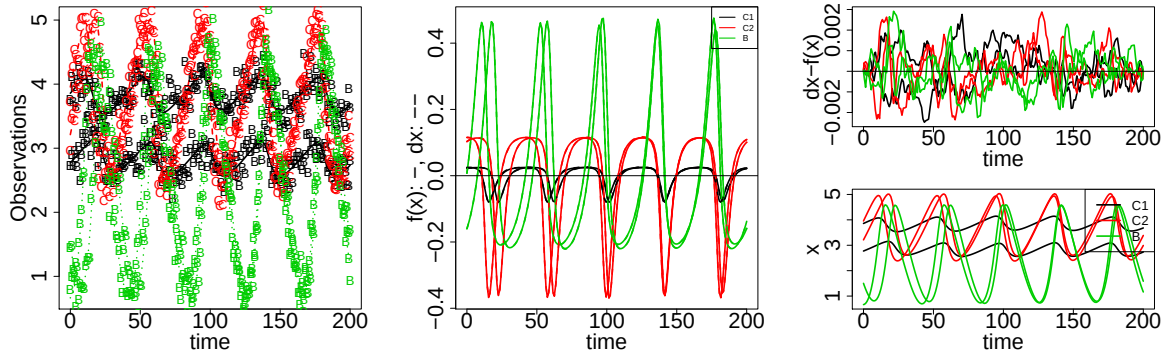

Figure 13: Diagnostic plots for repeated Rosenzweig-MacArthur simulations.

There are many reasons why this is not particularly satisfactory: we may wish to have parameters specific to each experiment, the experiments may have very different observation times, and we may want to produce our own diagnostic plots. For that the reader is encouraged to understand the structure of `lik` and `proc` objects so that they may put them together for themselves as outlined in Section 5.

5. User-defined fitting criteria and other extensions

The previous sections of this paper were focussed on functions that will allow using the profiling methodology quickly and with a minimum set-up cost. However, they can only be used with objective functions based on squared error and with certain structures for your data. Therefore, some of the functions employ defaults that may not be optimal for your system and data. **CollocInfer** allows profiling in a much more flexible class of models, but employing such flexibility requires more work on the part of the user.

In this package we allow the user to choose other measures to define the quality of fit for both terms. In the lower or inner optimization step, where θ is fixed, $C(\theta)$ minimizes

$$J(C|\theta) = \sum_{j \in C_O} \sum_{i=1}^{n} w_{ij} F\left[y_{ij}, \mathbf{c}_j \Phi(t_i)\right] + \sum_{j=1}^{d} \lambda \int P\left[D\mathbf{c}_j \Phi(t), f_j(\mathbf{C}\Phi(t), \boldsymbol{\theta})\right] \, dt. \tag{12}$$

while θ is then chosen to minimize

$$H(\theta) = \sum_{j \in C_O} \sum_{i=1}^{n} w_{ij} F\left[y_{ij}, \mathbf{c}_j(\theta) \Phi(t_i)\right]. \tag{13}$$

This formulation allows us to use the following measures of fit:

1. Fit function F is a measure of fit to the data. It will often be the sum of negative log density function values $F(y_i) = -\sum_j \log p_j(y_i | x_j(t_i), \theta)$. In this case, $F(y_i)$ is the *negative log likelihood* of the observations associated with the ith time value, and the sum over i is the *total negative log likelihood* of the data. Note that we only sum over the state variables that are observed.

2. Fit function P in the second term that quantifies failure to fit the differential equation, and represents a negative log likelihood for $\mathbf{x}$ if it is thought of as being generated by a random process. More general roughness measures using higher order derivatives or spatial co-ordinates are allowed in **CollocInfer**.

Notice that in both F and P we are no longer obliged to define fit in terms of the size of a difference; we could, for example, use differences between logarithms, differences between other transforms, or make no use of differences at all.

In **CollocInfer**, the `lik` and `proc` objects are designed to allow the evaluation of F and P respectively along with derivatives with respect to parameters θ and coefficients $\mathbf{C}$. Functions to evaluate these must be set up by the user and Section 8 details how they should be specified and some utilities that provide some shortcuts along with an example.

6. The Hénon map: A discrete system

Another flexibility that this framework allows is demonstrated in the next section. We detail the use of **CollocInfer** for discrete time difference equations:

$$\mathbf{x}(t+1) = \mathbf{f}(t, \mathbf{x}; \theta)$$

for $t = 1, \ldots, T$. The `discrete` option in the `Profile.LS` function sets up a basis expansion for such systems in which $\phi_j(t) = I(t = j)$. That is the jth row of $\mathbf{C}$ exactly records the values of the state variable $\mathbf{x}(j)$ at time $t = j$. If we make the unusual specification that $D\phi_j(t) = I(t = j + 1)$, then $\text{ISSE}(\theta; t, \mathbf{x})$ measures $\sum_{j=1}^{d} \sum_{t=1}^{T-1} [\mathbf{x}(t+1) - f_j(\mathbf{x}(t); t, \theta)]^2$.

In order to demonstrate this option, we employ the Hénon map:

$$x_{i+1} = 1 - a x_i^2 + y_i$$
$$y_{i+1} = b x_i$$

Parameter values for a and b that yield chaotic behavior are:

```
R> hpars <- c(1.4, 0.3)
```

First, we generate some data, leaving off the first 20 time points as transients

```
R> ntimes <- 200
R> x <- c(-1, 1)
R> X <- matrix(0, ntimes + 20, 2)
R> X[1,] <- x
R> for (i in 2:(ntimes + 20)) {
+    X[i,] <- make.Henon()$ode(i, X[i - 1, ], hpars, NULL)
+  }
R> X <- X[20 + 1:ntimes, ]
R> Y <- X + 0.05 * matrix(rnorm(ntimes * 2), ntimes, 2)
R> t <- 1:ntimes
```

This model creates a complex pattern, as can be seen in Figure 14.

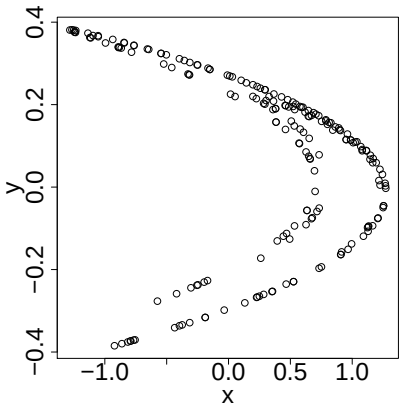

Figure 14: Data from the Henon Map.

In the generalized profiling framework, the Hénon map model is

$$\mathbf{x}_{t+1} = \mathbf{f}(\mathbf{x}_t, \theta) + \boldsymbol{\epsilon}_i$$

and we smooth the data Y to create a time-series $\hat{\mathbf{x}}_t(\theta)$ via the analogous idea to smoothing-based models for differential equations. More specifically, we re-define the penalty ISSE to be

$$\text{ISSE}(\theta, \mathbf{x}) = \sum_{t=1}^{n-1} \sum_{j=1}^{d} w_{tj} \left(x_{t+1,j} - f_j(\mathbf{x}_t, \theta)\right)^2$$

To fit within the profiling paradigm, we have to employ a basis expansion. In this case we just use an indicator function for the current time step being time k: $\phi_k = I(t == k)$ so that the coefficient matrix C gives exactly the value of $\mathbf{x}_i$ on each row t. Equivalent to evaluating the derivative of $\mathbf{x}$, we just need to evaluate $\mathbf{x}$ at the next time point $D\phi_k = I(t == k + 1)$ so that ISSE takes the form as previously defined.

We can now proceed with the generalized profiling recipe: minimizing PENSSE$(\mathbf{x}, \theta)$ to estimate $\mathbf{x}_t(\theta)$ and then minimizing squared error to estimate θ. As we have seen above, we do not need to restrict ourselves to squared error. To carry out the procedure in **CollocInfer**, we still employ the `LS.setup` functions, by using the `discrete = TRUE` option to specify that we consider a discrete-time dynamical system. We set starting values from perturbed parameters

```
R> hpars2 <- c(1.3, 0.4)
```

select a smoothing parameter

```
R> lambda <- 10000
```

and use the observations as our first guess at coefficients

```
R> coefs <- as.matrix(Y)
```

The `make.Henon()` function creates the list of functions to evaluate the Hénon map and its derivatives. Here, using the `discrete = TRUE` option indicates to **CollocInfer** that a discrete-time system is being modeled:

```
R> profile.obj <- LS.setup(pars = hpars2, coefs = coefs, fn = make.Henon(),
+    basisvals = NULL, lambda = lambda, times = t, discrete = TRUE)
R> lik <- profile.obj$lik
R> proc<- profile.obj$proc
R> Ires1 <- inneropt(data = Y, times = t, pars = hpars2, coefs, lik, proc,
+    in.meth = "nlminb", control.in)
R> Ores1 <- outeropt(data = Y, times = t, pars = hpars2, coefs = coefs,
+    lik = lik, proc = proc, in.meth = "nlminb", out.meth = "nlminb",
+    control.in = control.in, control.out = control)
R> parest <- Ores1$pars
```

The estimated parameters are:

```
R> parest
```

```
[1] 1.4066982 0.3010085
```

In order to plot the results, it is helpful to use separate panels for each dimension of $\mathbf{x}$ because the discrete-time system looks very rough as a time-series. The following code produces, in order, a comparison of $\hat{\mathbf{x}}_t(\theta)$ with the data, a comparison of $\hat{\mathbf{x}}_{t+1}(\theta)$ with $\mathbf{f}(\hat{\mathbf{x}}_t(\theta), \theta)$ and a scatterplot of $\hat{\mathbf{x}}_{t+1}(\theta) - \mathbf{f}(\hat{\mathbf{x}}_t(\theta), \theta)$ (see Figure 15) replicated from

```
R> Fit <- Ores1$coef
R> par(mfrow = c(2, 1))
R> plot(t, Fit[, 1], type = "l", ylab = "x", xlab = "time")
R> points(t, Y[, 1], col = 2)
R> plot(t, Fit[, 2], type = "l", ylab = "x", xlab = "time")
R> points(t, Y[, 2], col = 2)
```

we also compare prediction and new estimate – equivalent of derivative plots

```
R> pred <- Ores1$proc$more$fn(t, Fit, Ores1$pars, Ores1$proc$more$more)
R> par(mfrow = c(2, 1))
R> plot(t, pred[, 1], type = "l", ylab = "x", xlab = "time")
R> points(t[-ntimes], Fit[-1,1], col = 2, type = "l")
R> plot(t, pred[, 2], type = "l", ylab = "x", xlab = "time")
R> points(t[-ntimes], Fit[-1,2], col = 2, type = "l")
```

and look at the deviation between them

```
R> matplot(t[-ntimes], Fit[-1, ] - pred[-ntimes, ], xlab = "time",
+    ylab = "Model Residuals", pch = c("x", "y"))
```

To demonstrate that alternatives to squared error can be employed, we consider a simple variant using the multivariate normal log likelihood for transitions with a simple re-scaling of the variance of the $\boldsymbol{\epsilon}$:

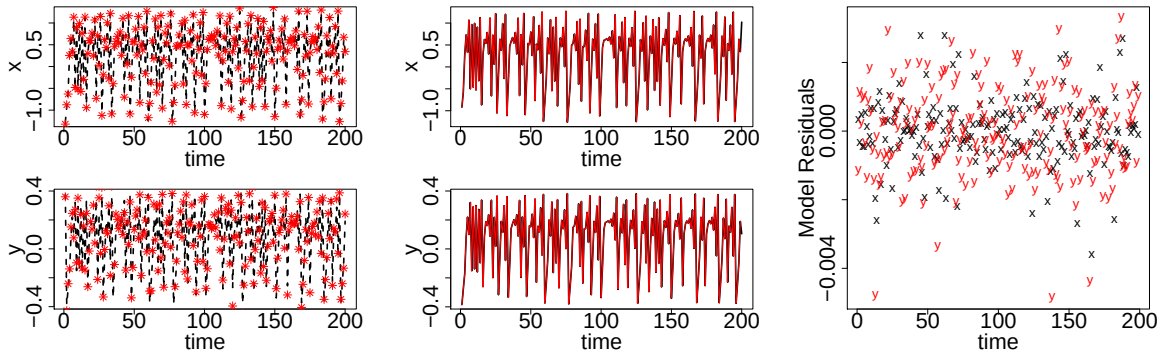

Figure 15: Diagnostic plots from fitting the Hénon map.

```
R> var <- c(1, 0.01)
R> Ires3 <- Smooth.multinorm(make.Henon(), data = Y, t, pars = hpars2, coefs,
+    basisvals = NULL, var = var, in.meth = "nlminb",
+    control.in = control.in, discrete = TRUE)
R> Ores3 <- Profile.multinorm(fn = make.Henon(), data = Y, times = t,
+    pars = hpars2, coefs = coefs, basisvals = NULL, var = var,
+    fd.obj = NULL, more = NULL, quadrature = NULL,
+    in.meth = "nlminb", out.meth = "optim", control.in = control.in,
+    control.out = control.out, eps = 1e-6, active = NULL,
+    discrete = TRUE)
```

The resulting estimates are:

```
R> Ores3$pars

[1] 1.4059512 0.3003313
```

7. The MATLAB version of CollocInfer

A parallel version of the **CollocInfer** package is maintained in the MATLAB language and is, at the time of writing, also distributed within the R package.

Both versions of the package are written so as to maintain their names and functionality as closely as possible within the syntax of their respective languages. The most noticeable change within the **CollocInfer** naming conventions is the replacement of '`.`' in R with '`_`' in MATLAB within variable and function names. Without providing an exhaustive list of notational differences we note that some are particularly relevant to the functionality presented here:

- List objects in R are replaced by `struct` objects in MATLAB in which the named elements are accessed by the `.` operator rather than `$`. Thus one access `lik.fn` rather than `lik$fn`.
- Bracketing and indexing conventions differ with `coefs[, 1]` in R being replaced by `coefs(:,1)` in MATLAB.

- MATLAB does not employ named dimensions so that in the FitzHugh-Nagumo example, `x[, "V"]` must be replaced by `x(:,1)`. Higher dimensional arrays are indexed by further indices in each language.
- An important detail is that the default behavior of each language when faced with the extraction of a particular dimension in an array. For example, if `x` is $n \times p$, R will treat `x[, 1]` as a vector, removing the singleton dimension whereas MATLAB will treat `x(:, 1)` as a matrix of dimension $n \times 1$. Each default can produce errors, although we have found that `R`'s choice is more likely to do so in **CollocInfer**.
- Functions in MATLAB may return multiple arguments whereas R requires that these be concatenated into a list so that MATLAB's call to `outeropt` appears as:

  ```
  [pars_opt, ncoefs, value, gradient] = ...
          outeropt(times, data, coefs, allpars, lik, proc, active)
  ```

  where if we do not wish to record `value`, we can replace the syntax with `[]`. In the command above, `active` could also be replaced by `[]`. Note that the ordering of arguments in MATLAB functions is important, whereas they can be named in R as is the ordering of outputs.

A more detailed discussion of notational differences in the two languages can be found in Ramsay *et al.* (2009). Many of these syntactical differences can be picked up by opening a `.R` file in the MATLAB editor which will highlight syntax errors.

We will briefly examine some differences in performance between the two implementations. In general, we find that the MATLAB implementation of **CollocInfer** provides a speed-up of a factor around 10 on any given task. This is due to the implementation of both optimization routines and sparse matrix calculations. It should be noted, however, that for numerically solving ODE's, the **deSolve** package (Soetaert *et al.* 2010) in R has proved to be considerably more accurate than the `ode45` function in MATLAB. MATLAB also requires a license, which can be quite expensive, and does not provide access to the plethora of statistical modeling packages available in R.

8. Low-level details

This section describes the structure of `lik` and `proc` objects and the ways in which they can be constructed to incorporate a wide range of objective criteria and potential extensions of profiling methods. These are demonstrated at the end of this section by low-level code that produces these objects for our Rosenzweig-MacArthur model. As an initial step, below we demonstrate how right-hand side functions work.

8.1. Right-hand functions $f_j(t, \mathbf{x}|\theta)$ and their derivatives

A differential equation is defined by the right sides of the differential equations $f_j(t, \mathbf{x}, \theta), j = 1, \ldots, d$. We will refer to these functions as *right-hand functions* in this paper. In the FitzHugh-Nagumo equations (3), the right-hand functions are $f_1(\mathbf{x}, \theta) = c(V - V^3/3 + R)$ and $f_2(\mathbf{x}, \theta) = (V - a + bR)/c$, where $\mathbf{x} = (V, R)^\top$ and $\theta = (a, b, c)^\top$. (Note: These equations are a trifle unusual in that there is no dependency on t except through the state vector $\mathbf{x}(t)$.)

The user must supply a set of R functions that evaluate the values of certain mathematical functions and their derivatives at times of observation. We begin with the R functions that are associated with the right-hand functions $f_j(t, \mathbf{x}|\theta)$. These functions are required to run **CollocInfer**, but we will see that they can be replaced with generic functions based on finite differencing so that, at the risk of adding numerical error, the user only need provide the original right-hand function.

The functions that evaluate right-hand functions and their derivatives are supplied to **CollocInfer** in a named list object, and the names used for these list members must be as shown below in typewriter font in order for the **CollocInfer** functions to be able to locate these functions. Of course, other named list members may also be included for purposes specific to an application, but the following named list members are required.

`fn:` calculates the value of each of d right-hand functions at the n times of observation. This function returns an $n \times d$ matrix of values.

`dfdx:` calculates the n values of the derivative of each right-hand function with respect to the states. This function returns an $n \times d \times d$ array.

`dfdp:` calculates the n values of the derivative of each right-hand function with respect to parameters. This function returns an $n \times d \times p$ array where p is the length of the parameter vector θ.

`d2fdx2:` calculates the n values of the second derivatives with respect to states. This function returns an $n \times d \times d \times d$ array.

`d2fdxdp:` calculates the n values of the cross derivatives of each right-hand function with respect to state and parameters. This function returns an $n \times d \times d \times p$ array.

Each of these five functions has four arguments:

`times:` either a vector of times of observations, or a vector of quadrature points depending on which **CollocInfer** function is calling the function.

`x:` a matrix of process values corresponding to the times argument. The matrix has d columns corresponding to state variables and n rows corresponding to times. This argument contains state values $x(t)$ rather than data values, and therefore all columns should contain only numeric data.

`p:` a vector of length p of parameter values.

`more:` an optional argument containing any other information required to compute the results. Of particular importance are any constant or functional input variables with values $u_\ell(t)$, called forcing terms. In the event that a variety of types of additional input are required, the `more` object will be a list object.

It is wise to include argument checks at least at the `fn` function. For example, the number of columns in argument `x` should be equal to the number of state variables that the function is designed to handle; and checks may be needed for the contents of the `more` argument as well. Also, if there is the possibility of certain derivatives taking inadmissible values such as `Inf` or zero, this also should be checked.

For a useful example, function `make.fhn()` produces a list with exactly these members defined for the FitzHugh-Nagumo equations above. It constructs the following functions and places them in a list with entries

```
R> fn <- function(times, y, p, more) {
+    r <- y
+    dimnames(r) <- dimnames(y)
+    r[, "V"] <- p["c"] * (y[, "V"] - y[, "V"]^3 / 3 + y[, "R"])
+    r[, "R"] <- - (y[, "V"] - p["a"] + p["b"] * y[, "R"]) / p["c"]
+    return(r)
+  }
R> dfdx <- function(times, y, p, more) {
+    r <- array(0, c(dim(y), 2))
+    r[, "V", "V"] <- p["c"] - p["c"] * y[, "V"]^2
+    r[, "V", "R"] <- p["c"]
+    r[, "R", "V"] <- - 1 / p["c"]
+    r[, "R", "R"] <- - p["b"] / p["c"]
+    return(r)
+  }
R> dfdp <- function(times, y, p, more) {
+    r <- array(0, c(dim(y), length(p)))
+    dimnames(r) <- list(NULL, colnames(y), names(p))
+    r[, "V", "c"] <- (y[, "V"] - y[, "V"]^3 / 3 + y[, "R"])
+    r[, "R", "a"] <- 1 / p["c"]
+    r[, "R", "b"] <- - y[, "R"] / p["c"]
+    r[, "R", "c"] <- (y[, "V"] - p["a"] + p["b"] * y[, "R"]) / (p["c"]^2)
+    return(r)
+  }
R> d2fdx2 <- function(times, y, p, more) {
+    r <- array(0, c(dim(y), 2, 2))
+    dimnames(r) <- list(NULL, colnames(y), colnames(y), colnames(y))
+    r[, "V", "V", "V"] <- - 2 * p["c"] * y[, "V"]
+    return(r)
+  }
R> d2fdxdp <- function(times, y, p, more) {
+    r <- array(0, c(dim(y), 2, length(p)))
+    dimnames(r) <- list(NULL, colnames(y), colnames(y), names(p))
+    r[, "V", "V", "c"] <- 1 - y[, "V"]^2
+    r[, "V", "R", "c"] <- 1
+    r[, "R", "V", "c"] <- 1 / p["c"]^2
+    r[, "R", "R", "b"] <- - 1 / p["c"]
+    r[, "R", "R", "c"] <- p["b"] / p["c"]^2
+    return(r)
+  }
```

8.2. Finite differencing methods for estimating partial derivatives

Writing functions to evaluate all the derivatives required above can require a lot of coding

effort if the differential equations either have a large number of states or involve complicated right-hand functions f_j. Based on our experience, it is strongly suggested that symbolic computation software be used for even seemingly simple systems, and that the programmer also run checks on these functions before using them in **CollocInfer**.

As an option, **CollocInfer** employs a finite differencing scheme on the columns of the `fn` element. Even if the intention is to program the computation of the necessary partial derivatives, the use of finite differencing can be a helpful debugging tool in the early stages of an analysis.

Finite difference approximations to partial derivatives can be obtained by calling function `make.findif.ode()` which creates a list with members `fn`, `dfdx`, `dfdp`, `d2fdx2`, `d2fdxdp` that calculate derivatives by finite differencing.

This is a situation where having the `more` member is useful. In this case the `more` list should itself contain members

`fn:` the function whose derivatives are to be approximated by differencing.

`eps:` the time interval δt to be used in the differencing.

`more:` any further objects to be passed to link function.

Thus, for example, `dfdx` is written as

```
R> findif.ode$dfdx <- function(times, x, p, more) {
+    fx <- more$fn(times, x, p, more$more)
+    x <- array(0, c(dim(fx), ncol(x)))
+    for (i in 1:ncol(y)) {
+      tx <- x
+      tx[, i] <- x[, i] + more$eps
+      dfx[,,i] <- (more$fn(times, tx, p, more$more) - fx) / more$eps
+    }
+    return(x)
+  }
```

Here `more$fn` in the first statement is the `fn` object that defines the right-hand functions (e.g., the original `fhn.fn`). For each column of `x`, we increment the whole column by `more$eps` and use this to calculate finite differences in each value of the column. This makes use of R's vectorization and is very fast. The other member functions in `findif.ode` behave similarly.

By default we set `eps <- 1e-6`, but the user should be aware that this can introduce numerical errors, and especially if the values of right-hand functions f_j vary greatly from one variable to another. It is sound practice here as well as elsewhere in computational methodology to attempt to scale variables so as to keep their values within an order of magnitude of one another. This is one reason we used the log transformation in the chemostat example.

8.3. Cascading `more` arguments for layered functions

The example in the previous section demonstrates how the `more` argument is used to compose functions. We need the original `fn` right-hand functions, but it is called *within* the finite differencing functions and is passed to them as a member of the list in the `more` argument.

This type of passing can be used, for example, to specify a log transformation within finite differencing and we will use this strategy extensively. For example, we do this if we wish to use the transform $\mathbf{z} = \log \mathbf{x}$ to fit dynamics on the log scale along with finite differencing. To do this, we can write the function

```
R> logtrans.fun <- function(times, x, p, more) {
+    x <- exp(x)
+    dx <- more$fn(times, x, p, more$more)
+    dx <- dx / x
+    return(dx)
+  }
```

which requires `more$fn` to be our original right hand side function. This could be fit within a finite differencing method so that we have

```
R> rhs <- make.findif.ode()
R> rhs$more <- list(fn = logtrans.fun, eps = 1e-6)
R> rhs$more$more <- list(fn = RosMac2, more = NULL)
```

This will be employed further within the `lik` and `proc` objects that we define below. The system allows a very flexible composition of a lot of functions, but the user can get confused by the multiple nesting of `more` arguments and some careful thinking through may be necessary.

With these preliminaries, we continue to the definition of objects describing the fit to data and to process.

8.4. Defining `lik` objects for assessing data fit

The `lik` object contains a function to compute the value of the fit between observations and data. It also requires the coding of functions for evaluating the values of some of its partial derivatives. The `lik` object is a named list object with names for its members that must be exactly as shown below in order to allow the information associated with these names to be accessed by other functions in the **CollocInfer** package. Some of the names for these evaluator functions are also used for the list object containing the evaluator functions for the right-hand functions $f_j(t, \mathbf{x}, \theta)$ described above in Section 8.1. And they will also be used in the `proc` list object described below in Section 8.5, as well in other named list objects.

The required names of the list members and their contents for the `lik` named list object are:

`fn:` a function that calculates the data fit measure F for each of d states at n times t_i taken over the observed state variables, such as the negative log likelihood of the residuals. This function returns an n by d matrix.

`dfdx:` a function that calculates the partial derivatives of `fn` with respect to the values of the state variables $\mathbf{x}$. It returns an array of size $n \times d \times d$.

`dfdp:` a function that calculates partial derivatives with respect to parameters. It returns an array of size $n \times d \times p$, where p is the length of the parameter vector θ.

`d2fdx2:` a function that calculates second partial derivatives with respect to the values of the state variables $\mathbf{x}$. It returns an array of size $n \times d \times d \times d$.

`d2fdxdp:` a function that calculates cross partial derivatives for state variable values and parameters. It returns an array of size $n \times d \times d \times p$.

`bvals:` an $n \times K$ matrix giving the values of the basis at the observation times.

`more:` This is an optional member that contains any additional inputs that the functions may require. The `more` member is typically itself a named list with members that can be referenced by various functions described later in the paper.

The approximation of confidence regions for parameter estimates will also require that the user-defined functions in the `lik` object also contain the following partial derivatives with respect to the data argument `y`:

`dfdy:` a function that calculates the partial derivatives of `fn` with respect to data values. It returns a matrix of size $n \times d \times d$, but the values returned for the unobserved state values are not used.

`d2fdy2:` a function that calculates second partial derivatives with respect to the data values. It returns an array of size $n \times d \times d \times d$, but entries corresponding to unobserved variables are not used.

`d2fdxdy:` a function that calculates partial cross derivatives with respect to the data values and the process values. It returns an array of size $n \times d \times d \times d$, but entries corresponding to unobserved variables are not used.

The arguments for the evaluator functions `fn`, `dfdx`, `dfdy`, `dfdp`, `d2fdx2`, `d2fdxdy`, `d2fdy2`, `d2fdxp`, `d2fdydp` include those for the right-hand function evaluators $f_j(t, \mathbf{x}, \theta)$ described in Section 8.1 above, but they are augmented by a first argument `y` specifying the data and by a matrix of basis function values. That is, they are

`y:` a matrix or array of observation values. The first dimension is length n a the second dimension has length equal to the number of observed states.

`times:` a vector of n times of observations.

`x:` an n by d matrix of state values corresponding to the times values.

`p:` a vector of length p f parameter values.

`more:` an optional argument containing any other information required to compute the results.

These functions are returned in a named list object. The names for the members or entries in the list are the same as those for the right-hand functions.

While calculating the likelihood is fairly straightforward for most distributions, it can be somewhat more cumbersome to write down functions to calculate the four different derivatives. Therefore a number of constructor functions have been created to make these calculations easier. `lik` objects can be created by calls to `make...` functions that produce a list with the appropriate members. For each of these, the member `more` is required to have specific entries that are detailed below.

Error sum of squares function `SSElik`

The sum of squared errors criterion that is employed in the first sections of this paper, and often used as the default loss function, may also be viewed as the negative log likelihood for the Gaussian distribution. Function `make.SSElik()` creates a list with entries `fn`, `dfdx`, `dfdy`, `dfdp, d2fdx2`, `d2fdxdy`, `d2fdy2`, `d2fdxp`, `d2fdydp`. These functions calculate

$$l(\mathbf{y}_j, t_i, \mathbf{x}_j, p) = \sum_{j=1}^{d} w_{ij}(y_{ij} - f_j(t_i, \mathbf{x}_j, \theta))^2$$

and its corresponding derivatives. They require `more` to contain functions `fn`, `dfdx`, `dfdp`, `d2fdx2`, `df2dxdp` and `more$more` for further objects needed to evaluate the f_j. These functions take the arguments `t, x, p, more` which are the same as the corresponding entries in the `lik` constructions. However the function output should have an extra dimension. `fn` is a vector valued function and returns a $n \times d$ matrix. Similarly, `dfdp` returns an array of dimension $n \times d \times p$ and so forth. The dimensions for the array go in order: element of f, derivatives with respect to x, derivative with respect to p.

In addition, `more` should contain a member `weights`. This should be a matrix of the w_{ij} that is the same dimension as Y; however it can also be a vector of same length as either dimension as Y, in which case it is interpreted as being constant across the other dimension. `more` may also contain `names`, giving the names of the states, if these are used in `fn`. Similarly, it may contain `parnames` to give the names of the parameters.

Programmers interested in setting up their own `make...` functions might find `make.SSElik` useful as a prototype.

Multivariate normal loss function `make.multinorm`

This set of functions calculates a log multivariate normal distribution for each observation

$$l(\mathbf{y}, t, \mathbf{x}, p) = \frac{1}{2}(\mathbf{y} - \mathbf{f}(t, x, p))^\top V^{-1}(t, \mathbf{x}, p)(\mathbf{y} - f(t, \mathbf{x}, p)) - \frac{1}{2}\log\ \det(V(t, \mathbf{x}, p)).$$

This is a generalization of the `SSElik` functions to correlated processes whose correlation may vary over time and the state variables. These are most useful for defining `proc` functions.

Function `make.multinorm` returns a `lik` objects with `fn`, `dfdx, dfdy`, `dfdp`, `d2fdx2`, `d2fdxdy`, `d2fdy2`, `d2fdxp, d2fdydp` which calculate a multivariate normal distribution.

These functions require `more` to contain `fn`, `dfdx`, `dfdp, d2fdx2`, `df2dxdp` to calculate the f_j and their derivatives as in `SSElik`. It must also contain members `var.fn`, `var.dfdx`, `var.dfdp`, `var.d2fdx2`, `var.df2dxdp` to provide the same set of derivatives for $V(t, x, p)$. Since $V(t, x, p)$ is matrix-valued, the output of these functions must have an extra dimension; giving `var.dfdp` dimension $n \times d \times d \times p$, for example.

In addition, `more` contains entries `f.more` for additional objects to be passed to f and `v.more` contains additional objects to be passed to V. It may also contain `names`, giving the names of the states, if these are used in `fn` and `var.fn`. Similarly, it may contain `parnames` to give the names of the parameters.

Finite differencing with function `findif.loglik`

This function creates a straightforward finite difference approximation to the various required derivatives of the log likelihood. Since d and p are not expected to be overly large, this is not very computationally intensive unless the likelihood is itself expensive to evaluate. These functions are also helpful in providing a check for user-defined likelihood and derivatives.

Function `make.findif.loglik` produces a `lik` object with `fn`, `dfdx`, `dfdy`, `dfdp`, `d2fdx2`, `d2fdxdy`, `d2fdy2, d2fdxp`, `d2fdydp` to calculate the appropriate derivatives by fixed-step finite differencing. These functions require `more` to be a list containing entries:

`fn:` this is the function that `lik$fn` would use if derivatives were given to it explicitly.

`eps:` the discretization parameter to be used in the finite difference scheme.

`more:` any additional inputs to `more$fn`.

For example, setting

```
R> lik <- make.findif.loglik()
```

```
R> lik$more <- make.SSElik()$fn
```

results in a finite difference approximation to the squared error "negative log likelihood".

8.5. Defining `proc` objects for assessing equation fit

The `proc` object stores functions to calculate the second term, the roughness penalty P. This is structured in a similar manner to the `lik` objects. An examination of both the least squares version of the inner optimization criterion J in PENSSE$(\mathbf{x}, \theta)$ and its more general version in (12) indicates that we

- replace the summation over n discrete time points t_i by the integration over continuous t, and
- replace the noisy observed data values y_{ij} by the current derivative estimates dx_j/dt, which, like the data, are not expected to be exactly equal to their fitted values $f_j(t, \mathbf{x}, \theta)$.

An important difference is that while `lik$fn` returns a goodness of fit measure at each observation time point, `proc$fn` returns a single number evaluating the goodness of fit for the entire process. This is largely opaque in `inneropt` and `outeropt`, but allows the user to extend the **CollocInfer** functionality to spatial processes and processes that involve delays or integral-differential systems.

Quadrature points t_q and weights w_q

At the computational level, the integral of P is necessarily approximated by numerical quadrature. This involves a judicious discretization of t and replacing the integral by a summation over quadrature points t_q using quadrature weights w_q, so that

$$\int \left[D\mathbf{c}_j\Phi(t) - f_j(t, \mathbf{C}\Phi(t), \boldsymbol{\theta})\right]^2 \, dt \approx \sum_q^Q w_q \left[D\Phi(t_q)\mathbf{c}_j - f_j(\Phi(t_q)C, \boldsymbol{\theta})\right]^2 \tag{14}$$

in the least squares case and

$$\int P\left[D\mathbf{c}_j\Phi(t), f_j(t, \mathbf{C}\Phi(t), \boldsymbol{\theta})\right] dt \approx \sum_q^Q w_q P\left[D\Phi(t_q)\mathbf{c}_j, f_j(\Phi(t_q)C, \boldsymbol{\theta})\right] \tag{15}$$

in the more general setting (12).

Numerical quadrature plays an essential role in the collocation approach, or indeed in any method of approximating a solution to a differential equation. The user must supply define these quadrature points and weights. The reader is warned that more difficult dynamic systems involving sharp local curvatures in the solutions will demand a more sophisticated rules based on the expected behavior of the ODE.

However, when solutions have only mild curvatures, the quadrature points t_q may be equally spaced and sufficient in number to capture the required detail in the solution. The weights w_q may be then equal to $\delta = 1/(t_q - t_{q+1})$ everywhere except at the end points, where they will be $\delta/2$. This simple approach to quadrature is called the *trapezoidal rule*.

Defining the functions and their arguments

As in the definition of the `lik` object, the `proc` is a named list, some of the names are specifically required by the **CollocInfer** package, and the majority of these are user-defined functions. However, many useful defaults are provided.

The required names and their contents for the `proc` list are

`fn:` a function that calculates $P(d\mathbf{x}/dt, \mathbf{f}(\mathbf{x}; \theta))$; the regularization penalty (or log probability) of the process. This returns a scalar.

`dfdc:` a function that calculates the derivatives of `fn` with respect to coefficients in $\mathbf{C}$, returns a vector of length dK.

`dfdp:` a function that calculates the derivatives with respect to parameters in θ, and returns an vector of length p.

`d2fdc2:` a function that calculates the second derivatives with respect to coefficients, and returns a matrix of size $dK \times dK$.

`d2fdcdp:` a function that calculates the cross derivatives of coefficients and parameters, and returns a matrix of size $dK \times p$.

`more:` usually a named list object whose members provide additional information defining these functions. Two members that may optionally be provided are

`names:` d names for the state variables.

`parnames:` p names for the parameters.

`bvals:` a named list object defining the basis values and their first derivative values. The member names are:

`bvals:` a $Q \times K$ matrix of values of the basis functions at the quadrature points t_q.

`dbvals:` a $Q \times K$ matrix of values of the first derivatives of the basis functions at the quadrature points t_q.

Some applications may require members of list `bvals` containing higher derivatives of the basis functions at the quadrature points.

All of the functions above take the following arguments

`coefs:` the current estimate of the coefficients.

`bvals:` as given in the `bvals` member in `proc`.

`pars:` current parameter values.

`more:` a named list containing additional information that may be required. In particular, it must specify quadrature points and weights with names

`qpts:` a vector of length Q containing quadrature points t_q where the penalty is to be evaluated.

`weights:` a matrix of dimension $q \times d$ containing the quadrature weights w_q.

Named list more may also optionally have members

`names:` a list of d names for the state variables. These enable the functions defined above to state variables in terms of their names rather than indexes.

`parnames:` a list of p names for the parameters.

These rather general definitions for the `proc` functions imply somewhat more effort for the user in defining them. The payoff is a very general framework that can encompass both discrete and continuous time systems along with higher-order and spatial systems.

However, as with `lik` objects, a number of pre-defined functions have been constructed to create special `proc` objects. These may have different definitions for the `bvals` member of the `proc` list, as well as for the `more` member.

Error sum of squares `proc` *function* `make.SSEproc`

This is the analogue of the `make.SSElik` and is used by default, based on approximation to the integrated squared error version of the roughness penalty

$$P(\mathbf{C}, \theta) = \sum_{j=1}^{d} \sum_{q=1}^{Q} w_q \left[D\Phi(t_q)\mathbf{c}_j - f_j(t, \Phi(t_q)\mathbf{C}, \theta) \right]^2 .$$

The **CollocInfer** pre-specified function `make.SSEproc()` creates a `proc` named list with functional members `fn`, `dfdc`, `dfdp`, `d2fdc2` and `df2dcdp`. In addition, member `bvals` needs to be defined as a named list to hold

`bvals:` a $Q \times K$ array giving the values of the basis functions at the quadrature points.

`dbvals:` a $Q \times K$ array giving the values of the derivatives of the basis functions at the quadrature points.

The named list `more` should hold

`qpts:` a vector of quadrature points t_i where the penalty is to be evaluated.

`weights:` a matrix giving the quadrature weights w_{ij}.

Function `Cproc`

Function `Cproc` generalizes `SSEproc` to allow any log likelihood of $d\mathbf{x}/dt$ given $\mathbf{x}$:

$$P_j(\mathbf{C},\theta) = \sum_{q=1}^{Q} w_q l\left[D\Phi(t_q)\mathbf{c}_j, f_j(t,\Phi(t_q)\mathbf{C},\theta)\right]$$

It can be used to take any of the negative log likelihoods defined for `lik` objects and convert them into the corresponding `proc` objects, provided the derivatives with respect to y are defined in the `lik` object.

Function `SSEproc` is equivalent to calling

```
R> proc <- make.Cproc()
R> proc$more <- make.SSElik()
```

and defining `proc$bvals` and `proc$more$more` appropriately. However, `SSEproc` creates a useful shortcut.

Function `Dproc`

The `Dproc` functions provide similar functionality to `Cproc` but for discrete-time processes. These calculate

$$P_j = \sum_{i=1}^{N-1} l\left[\mathbf{c}_j\Phi(t_{i+1}), f_j(t,\mathbf{C}\Phi(t_i),\theta)\right].$$

We often define Φ to be a sequence of square functions with breaks in the midpoints between the t_i. This is then equivalent to estimating the discrete state of the system.

The arguments for function `Dproc` are mildly different:

`bvals:` contains a single $n \times K$ array giving the basis values at the n evaluation times. For a saturated basis, $n = K$ and in this case is just the identity matrix of order n.

`more$qpts` is an $n - 1$ vector of times.

The `proc` named list defined by function `Dproc` also requires the same members `fn`, `dfdx`, `dfdy`, `dfdp`, `d2fdx2`, `d2fdxdy`, `d2fdy2`, `d2fdxp`, `d2fdydp`, and `more` as required by function `Cproc`, to be held in `more`, which can be called by any of the `lik` functions.

Note that `Dproc` is the same as defining

```
R> bvals$bvals <- basisvals[1:(nrow(basisvals) - 1), ]
R> bvals$dbvals <- basisvals[2:nrow(basisvals), ]
R> more$qpts <- more$qpts[1:(nrow(basisvals) - 1)]
```

and using `Cproc` (`SSEproc` may also be used if `more$weights` is also changed to be $(Q-1)\times d$). However, `Dproc` has been included to make the distinction between discrete and continuous time systems.

8.6. Manual set-up of the Rosenzweig-MacArthur model

In order to put all this in context, we will give the code here that will produce the `lik` and `proc` objects for modeling the Rosenzweig-MacArthur model example above. Here, instead of using the `LSsetup` function to generate the `lik` and `proc` object, we go through the process step by step.

We assume that we have defined `RosMac2`, `RMbasis` and `data`, `RMpars`, `RMParnames` and `RMVarnames`. In other words, we assume that the code associated with the Rosenzweig-MacArthur model has already been run and we are just manually producing the `lik` and `proc` objects here.

We begin by creating matrices of the evaluation of the basis functions at the time points we wish to use. By only insisting on the values of the basis, we allow the user to employ their own set of basis functions as desired. We need the values of the basis system at observation time points:

```
R> bvals.obs <- eval.basis(time, RMbasis)
```

and quadrature times. Here we use the midpoints between breaks plus the end points The quadrature weights are all equal, but we have multiplied them by the λ that we are using:

```
R> qpts <- c(breaks[1], breaks[1:(length(breaks) - 1)] + diff(breaks) / 2,
+    breaks[length(breaks)])
R> qwts <- 1e5 * matrix(1, length(qpts), 3) / length(qpts)
```

For `proc` objects, the basis values are a list containing two matrices providing the values of the basis functions at the quadrature points, and the values of the derivatives of the basis functions:

```
R> bvals.quad <- list(bvals = eval.basis(qpts, RMbasis),
+    dbvals = eval.basis(qpts, RMbasis))
```

With these defined, we create the `lik` object. The command

```
R> lik.m <- make.SSElik()
```

sets up the squared error criterion for the data. This is a list of functions and their derivatives, to which we attach the values of the basis expansion at the observation time points with the command

```
R> lik.m$bvals <- bvals.obs
```

We now need to specify the transformation of the state variables that is to be compared with the data and functions to calculate various derivatives. In this case, we use finite differencing to compute the derivatives that we need; we can achieve this by first employing

```
R> lik.m$more <- make.findif.ode()
```

and then telling `findif.ode` that the function it is finite differencing is

```
R> lik.m$more$more <- list(fn = RMobsfn, eps = 1e-6, more = NULL)
```

along with the difference increment and the fact that no further inputs are needed.
We also need to give `lik.m` a set of weights, which in this example are all ones. This has to occur in the `more` member of `lik.m` because it is used inside `lik.m$fn`:

```
R> lik.m$more$weights <- array(1, dim(data))
```

We can also create the `proc` object manually. First we set up the squared error functions

```
R> proc.m <- make.SSEproc()
```

We also specify the basis values and their derivatives at the quadrature points

```
R> proc.m$bvals <- bvals.quad
```

Now we need to tell `SSEproc` about the right hand side of the ODE and its derivatives. Here we will use finite differencing again, as in `lik.m`:

```
R> proc.m$more <- make.findif.ode()
```

In fact, we are going to finite difference the right hand side of the the ODE for the log transformed data. Here we specify the log transform, along with the finite difference increment:

```
R> proc.m$more$more <- list(fn = make.logtrans()$fn, eps = 1e-6)
```

and then give it the (non log-transform) Rosenzweig-MacArthur equations:

```
R> proc.m$more$more$more$fn <- RosMac2
```

Note here that `make.logtrans()` produces a list of functions to evaluate various derivatives. Since this is being done by finite differencing, we only need the `fn` member of these. We could have reversed this order – used `make.logtrans()` first and then called `make.findif.ode()` for its `more` member. However, we have found that if you are employing finite differencing, it is best to do this as the last step rather than within other transformations.
Member `proc.m$more` also needs to include some members for internal processing. In particular the following specify the quadrature points and weights

```
R> proc.m$more$weights <- qwts
R> proc.m$more$qpts <- qpts
```

We will specify the parameter and variable names:

```
R> proc.m$more$parnames <- RMParnames
R> proc.m$more$names <- RMVarnames
```

At this point `lik.m` and `proc.m` are identical to `lik2` and `proc2` in Section 4.2 and the analysis there can be re-run with these to produce the same results.

Acknowledgments

This software was developed as part of the *Unifying Approaches to Statistical Inference in Ecology* working group at the National Center for Ecological Analysis and Synthesis. It was also supported by NSF Grants DEB-1353039 and DMS-1053252 and Federal Formula Funds Hatch Grant NYC-150446. Luo Xiao's work was supported by Grant Number R01NS060910 from the National Institute of Neurological Disorders and Stroke.

References

Becks L, Ellner SP, Jones LE, Hairston NG (2010). "Reduction of Adaptive Genetic Diversity Radically Alters Eco-Evolutionary Community Dynamics." *Ecology Letters*, **13**, 989–997. doi:10.1111/j.1461-0248.2010.01490.x.

Bellman R, Roth RS (1971). "The Use of Splines with Unknown End Points in the Identification of Systems." *Journal of Mathematical Analysis and Applications*, **344**, 26–33. doi:10.1016/0022-247x(71)90154-5.

Bock HG (1983). "Recent Advances in Parameter Identification Techniques for ODE." In P Deuflhard, E Harrier (eds.), *Numerical Treatment of Inverse Problems in Differential and Integral Equations*, pp. 95–121. Birkhäuser, Basel.

Brunel NJB (2008). "Parameter Estimation of ODE's via Nonparametric Estimators." *Electronic Journal of Statistics*, **2**, 1242–1267. doi:10.1214/07-ejs132.

Campbell DA, Hooker G, McAuley KB (2012). "Parameter Estimation in Differential Equation Models with Constrained States." *Journal of Chemometrics*, **26**, 322–332. doi:10.1002/cem.2416.

Cao J, Ramsay JO (2009). "Linear Mixed Effects Modeling by Parameter Cascading." *Journal of the American Statistical Association*, **105**, 365–374. doi:10.1198/jasa.2009.tm09124.

Cao J, Ramsay JO, Fussmann G (2008). "Estimating a Predator-Prey Dynamical Model with the Parameter Cascades Method." *Biometrics*, **64**, 959–967. doi:10.1111/j.1541-0420.2007.00942.x.

Deuflhard P, Bornemann F (2000). *Scientific Computing with Ordinary Differential Equations*. Springer-Verlag, New York. doi:10.1007/978-0-387-21582-2.

Ellner SP (2007). "Commentary on "Parameter Estimation in Differential Equations: A Generalized Smoothing Approach"." *Journal of the Royal Statistical Society B*, **16**, 741–796. doi:10.1111/j.1467-9868.2007.00610.x.

Ellner SP, Seifu Y, Smith RH (2002). "Fitting Population Dynamic Models to Time-Series Data by Gradient Matching." *Ecology*, **83**, 2256–2270. doi:10.1890/0012-9658(2002)083[2256:fpdmtt]2.0.co;2.

FitzHugh R (1961). "Impulses and Physiological States in Models of Nerve Membrane." *Biophysical Journal*, **1**, 445–466. doi:10.1016/s0006-3495(61)86902-6.

Fourer R, Gay DM, Kernighan BW (2003). *AMPL: A Modeling Language for Mathematical Programming*. Brooks/Cole – Thomson Learning, Pacific Grove.

Gugushvili S, Klaassen CAJ (2012). "$\sqrt{n}$-Consistent Parameter Estimation for Systems of Ordinary Differential Equations: Bypassing Numerical Integration via Smoothing." *Bernoulli*, **18**, 1061–1098. doi:10.3150/11-bej362.

Hodgkin AL, Huxley AF (1952). "A Quantitative Description of Membrane Current and Its Application to Conduction and Excitation in Nerve." *The Journal of Physiology*, **133**, 444–479. doi:10.1113/jphysiol.1952.sp004764.

Hooker G (2009). "Forcing Function Diagnostics for Nonlinear Dynamics." *Biometrics*, **65**, 928–936. doi:10.1111/j.1541-0420.2008.01172.x.

Hooker G, Biegler L (2007). "**IPOPT** and Neural Dynamics: Tips, Tricks and Diagnostics." *Technical Report BU-1676-M*, Department of Biological Statistics and Computational Biology, Cornell University. URL http://faculty.bscb.cornell.edu/~hooker/zebra_desc2.pdf.

Hooker G, Ellner SP, de Vargas Roditi L, Earn DJ (2011). "Parameterizing State-Space Models for Infectious Disease Dynamics by Generalized Profiling: Measles in Ontario." *Journal of the Royal Society Interface*, **8**, 961–974. doi:10.1098/rsif.2010.0412.

Hooker G, Xiao L, Ramsay JO (2016). ***CollocInfer***: *Collocation Inference for Dynamic Systems*. R package version 1.0.4, URL https://CRAN.R-project.org/package=CollocInfer.

King AA, Nguyen D, Ionides EL (2016). "Statistical Inference for Partially Observed Markov Processes via the R Package **pomp**." *Journal of Statistical Software*, **69**(12), 1–43. doi:10.18637/jss.v069.i12.

Körkel S (2002). *Das Softwarepaket **VPLAN***. URL http://ginger.iwr.uni-heidelberg.de/vplan/.

Ljung L (1995). "System Identification Toolbox for Use with MATLAB: User's Guide." SOP-95-89/SOP-96-68, URL http://opac.inria.fr/record=b1070642.

Nagumo JS, Arimoto S, Yoshizawa S (1962). "An Active Pulse Transmission Line Simulating a Nerve Axon." *Proceedings of the IRE*, **50**, 2061–2070. doi:10.1109/jrproc.1962.288235.

Newey WK, West KD (1987). "A Simple, Positive Semi-Definite, Heteroskdasticity and Autocorrelation Consistent Covariance Matrix." *Econometrica*, **55**, 703–708. doi:10.2307/1913610.

Qi X, Zhao H (2010). "Asymptotic Efficiency and Finite-Sample Properties of the Generalized Profiling Estimation of Parameters in Ordinary Differential Equations." *The Annals of Statistics*, **38**, 435–481. doi:10.1214/09-aos724.

Ramsay JO, Hooker G, Campbell D, Cao J (2007). "Parameter Estimation in Differential Equations: A Generalized Smoothing Approach." *Journal of the Royal Statistical Society B*, **65**, 741–796. doi:10.1111/j.1467-9868.2007.00610.x.

Ramsay JO, Hooker G, Graves S (2009). *Functional Data Analysis with R and MATLAB*. Springer-Verlag, New York. doi:10.1007/978-0-387-98185-7.

Ramsay JO, Silverman BW (2005). *Functional Data Analysis*. Springer-Verlag, New York. doi:10.1007/978-1-4757-7107-7.

Ramsay JO, Wickham H, Graves S, Hooker G (2014). ***fda**: Functional Data Analysis*. R package version 2.4.4, URL https://CRAN.R-project.org/package=fda.

R Core Team (2016). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Rosenzweig MP, MacArthur RH (1969). "Graphic Representation and Stability Conditions of Predator-Prey Interaction." *American Naturalist*, **97**, 209–223. doi:10.1086/282272.

Soetaert K, Petzoldt T, Setzer RW (2010). "Solving Differential Equations in R: Package **deSolve**." *Journal of Statistical Software*, **33**(9), 1–25. doi:10.18637/jss.v033.i09.

The MathWorks, Inc (2011). *MATLAB – The Language of Technical Computing, Version 7.12.635 (R2011a)*. The MathWorks, Inc., Natick. URL http://www.mathworks.com/products/matlab/.

Tjoa IB, Biegler LT (1991). "Simultaneous Solution and Optimization Strategies for Parameter Estimation of Differential-Algebraic Equation Systems." *Industrial Engineering and Chemical Research*, **30**, 376–385. doi:10.1021/ie00050a015.

Varah JM (1982). "A Spline Least Squares Method for Numerical Parameter Estimation in Differential Equations." *SIAM Journal on Scientific Computing*, **3**, 28–46. doi:10.1137/0903003.

Wächter A, Biegler LT (2006). "On the Implementation of an Interior-Point Filter Line-Search Algorithm for Large-Scale Nonlinear Programming." *Mathematical Programming*, **106**, 25–57. doi:10.1007/s10107-004-0559-y.

Wilkinson D (2013). ***smfsb**: Stochastic Modelling for Systems Biology, Second Edition*. R package version 1.1, URL https://CRAN.R-project.org/package=smfsb.

Wilson HR (1999). *Spikes, Decisions and Actions: The Dynamical Foundations of Neuroscience*. Oxford University Press, Oxford.

Wood S (2010). "Statistical Inference for Noisy Nonlinear Ecological Dynamic Systems." *Nature*, **466**(7310), 1102–1104. doi:10.1038/nature09319.

Wu H, Miao H, Warnes GR, Wu C, LeBlanc A, Dykes C, Demeter LM (2008). "DEDiscover: A Computation and Simulation Tool for HIV Viral Fitness Research." In *BMEI 2008 – International Conference on BioMedical Engineering and Informatics, 2008*, volume 1, pp. 687–694.

Wu H, Xue H, Kumar A (2012). "Numerical Discretization-Based Estimation Methods for Ordinary Differential Equation Models via Penalized Spline Smoothing with Applications in Biomedical Research." *Biometrics*, **68**, 344–52. doi:10.1111/j.1541-0420.2012.01752.x.

Affiliation:

Giles Hooker
1186 Comstock Hall
Cornell University
Ithaca, NY 14853, United States of America
E-mail: giles.hooker@cornell.edu
URL: http://faculty.bscb.cornell.edu/~hooker/

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
November 2016, Volume 75, Issue 2 *Submitted:* 2014-05-11
doi:10.18637/jss.v075.i02 *Accepted:* 2015-08-25