

# CollocInfer: An R Library for Collocation Inference for Continuous- and Discrete-Time Dynamic Systems

Giles Hooker, Luo Xiao and Jim Ramsay

May 7, 2010

## Abstract

This monograph details the implementation and use of R routines for smoothing-based estimation of continuous-time nonlinear dynamic systems. These routines represent an extension of the *generalized profiling* (GP) methods described in Ramsay et al. (2007) for estimating parameters in nonlinear ordinary differential equations (ODEs). It includes an interface to the R package `fda`. The package also supports discrete-time systems. We describe the methodological and computational framework and the necessary steps to use the software.

## Contents

<b>1</b>	<b>Overview of the CollocInfer package</b>	<b>3</b>
1.1	Some notation for the data and the model . . . . .	4
1.2	An example: The FitzHugh-Nagumo equations . . . . .	5
1.3	The collocation method and basis function expansions for $\mathbf{x}$ . . . . .	5
1.4	Parameter estimation by generalized profiling (GP) . . . . .	6
<b>2</b>	<b>Setting up the data for a CollocInfer analysis</b>	<b>8</b>
<b>3</b>	<b>Setting up the functions: Basic level using squared errors</b>	<b>9</b>
3.1	Functions for evaluating $f_i(t, \mathbf{x}, \boldsymbol{\theta})$ and its derivatives . . . . .	9
3.2	Computing derivatives by differencing: <code>make.findif.loglik</code> . . . . .	12
3.3	Setting up a first FitzHugh Nagumo example . . . . .	13
<b>4</b>	<b>Setting up functions for customized fit measures</b>	<b>16</b>
4.1	User-defined fit measures . . . . .	16
4.2	Defining <code>lik</code> objects to assess data fits . . . . .	17
4.2.1	<code>SSElik</code> : The ordinary least squares <code>lik</code> object . . . . .	18
4.2.2	<code>multinorm</code> : Generalized least squares data fits . . . . .	19
4.3	Defining <code>proc</code> objects for assess equation fits . . . . .	20

4.3.1	Quadrature points $t_q$ and weights $w_q$ for numerical integration	20
4.3.2	Defining the <code>proc</code> functions and their arguments	20
4.3.3	<code>SSEproc</code> : The ordinary least squares <code>proc</code> object	22
4.3.4	<code>Cproc</code> and <code>Dproc</code> : generalized least squares <code>proc</code> objects	22
4.4	Link functions $g$ for indirect data–model relations	23
4.4.1	<code>genlin</code>	25
4.5	Variance functions for defining generalized least squares fits	25
<b>5</b>	<b>Confidence intervals for estimates of parameters in <math>\theta</math></b>	<b>27</b>
<b>6</b>	<b>Optimizing functions and the functions they call</b>	<b>29</b>
6.1	Inner optimization of $J$ to estimate coefficients: <code>inneropt</code>	30
6.2	Outer optimization $H$ to estimate parameters: <code>outeropt</code>	31
6.3	Function for confidence intervals: <code>Profile.covariance</code>	34
6.4	A special purpose optimizer for least squares: <code>Profile.GausNewt</code>	35
6.5	Gradients and Hessians for least squares: <code>ProfileSSE</code>	36
6.6	Computing starting coefficient values: <code>FitMatchOpt</code>	37
6.7	Estimating parameters given $\mathbf{x}$ : <code>ParsMatchOpt</code>	39
<b>7</b>	<b>More functions for least squares estimation</b>	<b>41</b>
7.1	Setting up <code>lik</code> and <code>proc</code> objects: <code>LS.setup</code>	41
7.2	Smoothing the data given parameters: <code>Smooth.LS</code>	43
7.3	Least squares generalized profiling: <code>Profile.LS</code>	44
<b>8</b>	<b>Positive State Vectors</b>	<b>46</b>
8.1	Utility Functions	46
8.1.1	<code>logstate.lik</code>	46
8.1.2	<code>exp.Cproc</code>	46
8.1.3	<code>exp.Dproc</code>	46
8.1.4	<code>logtrans.ode</code>	46
<b>9</b>	<b>Estimation for the FitzHugh-Nagumo system</b>	<b>47</b>
9.1	Unreplicated Data	47
9.2	Replicated Observations	50
<b>10</b>	<b>Estimation for the groundwater system</b>	<b>52</b>
<b>11</b>	<b>The Hénon Map: A Discrete System</b>	<b>55</b>
<b>12</b>	<b>SEIR Equations and Positive State Vectors</b>	<b>57</b>
<b>13</b>	<b>Equations for Ecologies and Observations of Linear Combinations of States</b>	<b>63</b>
<b>14</b>	<b>Acknowledgements</b>	<b>67</b>

# 1 Overview of the CollocInfer package

The CollocInfer package implements smoothing-based approaches to estimating parameters in dynamic systems. Dynamic systems model the nonlinear behavior often found in real-world processes, and, because they involve either derivatives (continuous time) or differences (discrete time), they are fundamentally models for how the process changes. These systems are typically nonlinear and involve multiple variables. The systems can involve either continuous or discrete time, although for simplicity the notation used in the manual will mostly be for continuous time systems.

In mathematical notation, the CollocInfer package assumes an underlying real-world possibly  $d$ -dimensional multivariate process  $\mathbf{x}$  whose *state* at time  $t$  is the vector  $\mathbf{x}(t)$  of length  $d$ . The state is assumed to satisfy a set of ordinary differential equations

$$\frac{d}{dt}\mathbf{x}(t) = \mathbf{f}[t, \mathbf{x}(t), \boldsymbol{\theta}],$$

the  $i$ th of which is

$$\frac{d}{dt}x_i(t) = f_i[t, \mathbf{x}(t), \boldsymbol{\theta}], \tag{1}$$

The right sides of these equations, which we call in this manual *right-hand functions*, are defined by a set of known functions  $f_i, i = 1, \dots, d$ , that depend on the current value of potentially all of the state variables in  $\mathbf{x}$ . The right-hand functions may also depend on  $t$  in ways other than through  $\mathbf{x}(t)$ . For example, it is common to have right-hand functions of the form  $f_i(\mathbf{u}(t), \mathbf{x}(t), \boldsymbol{\theta})$  where the functions defining vector  $\mathbf{u}(t)$  represent external inputs to the dynamic systems which are often called *forcing functions*.

The right side of these equations  $\mathbf{f}_i[t, \mathbf{x}(t), \boldsymbol{\theta}]$  is also defined by a set of parameters contained in the parameter vector  $\boldsymbol{\theta}$  of length  $p$ . The primary goal of the package is to estimate these parameters.

The methods described represent an extension of the generalized profiling (GP) methods described in Ramsay, Hooker, Campbell, and Cao (2007) for estimating parameters in ordinary differential equations. It is not essential to have read this paper in order to use this library, since this manual aims to present the essential ideas in that paper with only as much technical detail as is necessary. The remaining subsections of this section provide this exposition. Users already sufficiently familiar with this material may want to skip to Section 2 where instructions on how to use the package begin.

The code builds on, and interfaces with the `fda` library for functional data analysis. For a full review of functional data analysis, see Ramsay and Silverman (2005), but a more applied and informal introduction intended for users of the R language is Ramsay, Hooker, and Graves (2009). Installation of the `fda` library is not required to use this library, but at least some understanding of functional data analysis is likely to be helpful.

In the remainder of this introduction we

- specify some notation for the dimensions of various vectors, matrices and arrays

- give a simple example of a dynamic system
- describe the collocation approach to estimating state functions  $x_i$ , and
- summarize the generalized profiling approach to parameter estimation.

The code is structured to be extendable to more general problems. In particular, this framework allows second and higher-order dynamics to be implemented naturally. Additionally, while our discussion of these methods is focussed for  $t$  taking real values for continuous time systems or integers for discrete-time systems, there is nothing inherent in the code that prevents the estimation of spatial-temporal processes, integro-differential systems and many more options.

## 1.1 Some notation for the data and the model

Bold-faced variables such as  $\mathbf{x}$  refer to vectors or vector-valued functions, and  $x_i$  refers to the  $i$ th element of  $\mathbf{x}$ . Capital letters refer to matrices or matrix-valued functions.

We will refer to the collection of observed data, including observation times, as  $Y$ , even when the data may not fit into matrix format. Specifically,  $y_{ij}$  indicates the observation at time  $t_j$  of state variable  $x_i$ , an observation that is usually subject to some error. For simplicity, we assume in this manual that the times of observation are common to all observed variables. We do not assume, however, that all state variables are observed, and when the  $i$ th variable is unobserved, the  $y_{ij}$ 's can be considered to all have the R “not available” or missing data value **NA**.

Variables in typewriter font, such as `p`, refer to variable names in the R code. For example, we use `p` to refer to the length of the parameter vector, but  $\boldsymbol{\theta}$  to refer to the parameter vector itself in the R code.

Here is a list of notation for various important constants such as the dimensions of the data.

$d$ : the dimension of state vector  $\mathbf{x}$ , which is the number of differential equations in the dynamic system.

$d_o$ : the number of state functions  $x_i$  that are observed. It is assumed that not all of the state functions are associated with observations, so that it is quite likely that that  $d_o < d$ . In that event, we use the  $D_o$  to indicate the set of indices  $i$  corresponding to observed variables.

$p$ : the dimension of the parameter vector  $\boldsymbol{\theta}$ .

$n$ : the number of times  $t_j$  at which a continuous time system is observed.

$N$ : the number of time steps in discrete-time systems.

$K$ : the number of basis functions. In practice, this is often the same as either  $Q$  or  $N$ .

$L$ : the number of discrete points in a discrete time system. Note that observations are not automatically taken at points corresponding these discrete points, so that  $L$  and  $n$  can be different.

$Q$ : the number of quadrature points to evaluate the integrals associated with continuous-time systems.

## 1.2 An example: The FitzHugh-Nagumo equations

The FitzHugh-Nagumo equations provide a simple but fairly representative test-bed for the understanding and applying the CollocInfer package. They are given by a two-variable ( $d = 2$ ) differential equation:

$$\begin{aligned}\frac{d}{dt}V &= c(V - V^3/3 + R) \\ \frac{d}{dt}R &= -(V - a + bR)/c.\end{aligned}\tag{2}$$

These equations are intended to capture the essential dynamic properties of electrical response of a neuron, which consists of a string of localized changes in voltage  $V$  across the membrane of the neuron. Variable  $R$  represents a sum of “recovery” ion currents. The parameters to be estimated are  $a$ ,  $b$  and  $c$  ( $p = 3$ ). State variable  $V$ , or  $x_1$  in general notation, is a measurable variable, but variable  $R$  (or  $x_2$ ) represents a collection of measurable variables, and is therefore not itself directly measurable. We would normally assume that we only have observations of  $V$ , but for purposes of illustration, we might pretend that data are available for  $R$  also.

## 1.3 The collocation method and basis function expansions for $\mathbf{x}$

The *generalized profiling* methodology uses the *collocation method* for the approximation of solutions of differential equations. The state of a system of  $d$  differential equations at time  $t$  is denoted in this manual by the  $d$ -dimensional vector  $\mathbf{x}(t)$ , and the collocation approach represents  $\mathbf{x}(t)$  as a basis expansion:

$$\mathbf{x}(t) = \Phi(t)\mathbf{C}\tag{3}$$

using a set of  $K$  basis functions whose values at time  $t$  are contained in vector  $\Phi(t) = (\phi_1(t), \dots, \phi_K(t))'$ , and  $K$  by  $d$  matrix  $\mathbf{C}$  contains in its columns coefficients of the basis function expansion of each variable. That is, the  $i$ th state at time  $t$  has the basis function expansion

$$x_i(t) = \Phi(t)\mathbf{c}_i = \sum_{k=1}^K c_{ik}\phi_k(t),\tag{4}$$

where  $\mathbf{c}_i$  is a vector of coefficients  $c_{ik}$  of length  $K$  contained in the  $i$ th column of  $\mathbf{C}$ .

This package uses B-spline basis functions for  $\phi_k, k = 1, \dots, K$ . B-spline basis functions are constructed by joining polynomial segments end-to-end at junction

points called *knots*. We again refer the reader who needs further explanation to references such as Ramsay et al. (2009).

Both the number and location of knots play critical roles in the success of a collocation analysis. It is fairly typical that dynamic systems exhibit sharp curvature over small time intervals, often associated with a sudden change in an input variable. There must be enough knots in the vicinity of these regions to accommodate the required curvature. A collocation analysis must often be conducted through several cycles, refining knot placement at each cycle to allow for curvature suggested by the results of the previous cycle. A common strategy is begin with a dense equally-spaced knot sequence, and over subsequent cycles to reduce the number of knots over intervals of mild curvature.

#### 1.4 Parameter estimation by generalized profiling (GP)

Here we describe the generalized profiling (GP) strategy for parameter estimation as presented in Ramsay et al. (2007). We here discuss GP in the more familiar context of minimizing error sum of squares measures of fit to the data and fit to the differential equation, but in Section 4 cover various extensions involving more general user-defined measures of fit, as well as other useful features.

The generalized profiling methodology, also called *parameter cascading*, is a two-level procedure involving a low-level or inner optimization step nested within a high level outer optimization. The functions being optimized are not the same for all levels; the lower level optimization involves a smoothed or regularized measure of fit, and the upper level is a more straightforward fit measure.

In the inner optimization, the parameters in vector  $\boldsymbol{\theta}$  are held fixed, and an inner optimization criterion  $J(\mathbf{C}|\boldsymbol{\theta})$  is optimized with respect to the coefficients in matrix  $\mathbf{C}$  only. In effect, this makes  $\mathbf{C}$  a function of  $\boldsymbol{\theta}$ , that is,  $\mathbf{C}(\boldsymbol{\theta})$ , since each time  $\boldsymbol{\theta}$  is changed in any way, it is necessary to repeat this inner optimization step. A function defined in this manner by optimizing a criterion is called an *implicit function*.

The lower or inner level fitting criterion is

$$J(\mathbf{C}|\boldsymbol{\theta}) = \sum_{i \in D_o} \sum_{j=1}^n w_{ij} [y_{ij} - \Phi(t_j)\mathbf{c}_i]^2 + \sum_{i=1}^d \lambda_i \int \left[ \frac{d}{dt} \Phi(t)\mathbf{c}_i - f_i(t, \Phi(t)\mathbf{C}, \boldsymbol{\theta}) \right]^2 dt \quad (5)$$

where  $\mathbf{c}_i$  is the  $i$ th row of coefficient matrix  $\mathbf{C}$ .

The first term in  $J$  measures how well the state function values  $x_i(t_j)$  fit the data  $y_{ij}$  in terms of a weighted error sum of squares. The summation over  $i$  is only over those state functions or processes that are actually observed.

The second term measures how closely each of the state functions satisfy the corresponding differential equation (1), expressed in terms of the integrated square of the difference between the right and left sides of (1). Here  $i$  ranges through all the functions  $x_i$ . This term, too, is a weighted sum of squares, but in this case the weights  $\lambda_i$  vary over  $i$ , but not over  $j$ . The summation over time index  $j$  in the first term is now replaced by an integration over  $t$ .

The *smoothing parameters*  $\lambda_1, \dots, \lambda_d$  arbitrate between these two competing types of fit. As a  $\lambda_i$  gets larger and larger, more and more emphasis is put on having  $x_i$  satisfy the differential equation, as opposed to fitting the data. Conversely, as  $\lambda_i$  goes to zero, so much emphasis is put on fitting the data that  $x_i$  will eventually fit the data points exactly, given enough basis functions. Smoothing parameters give us a valuable degree of control over which of these types of fit we wish to emphasize. In fact, it is usual to use a value of each  $\lambda_i$  that strikes a reasonable compromise, and to compare these results to those obtained as  $\lambda_i$  goes large enough to define a nearly exact solution to the differential equation.

The estimate for  $\boldsymbol{\theta}$  is determined at the higher or outer level, where the GP method computes those  $\boldsymbol{\theta}$ -values that minimize only the first data-fitting term

$$H(\boldsymbol{\theta}) = \sum_{i \in D_0} \sum_{j=1}^n w_{ij} [y_{ij} - \Phi(t_j) \mathbf{c}_i(\boldsymbol{\theta})]^2. \quad (6)$$

Note that  $\mathbf{c}_i(\boldsymbol{\theta})$  is a function of the parameters in  $\boldsymbol{\theta}$ , since, for any set of  $\boldsymbol{\theta}$ -values, criterion  $J(\mathbf{C}|\boldsymbol{\theta})$  defined in (5) is optimized with respect to the coefficients in  $\mathbf{c}_i$ . This is the key idea underlying the generalized profiling or parameter cascade algorithm (Cao and Ramsay, 2009).

Ramsay et al. (2007) demonstrated that as the  $\lambda_i \rightarrow \infty$  the estimated parameters converge on those that would be estimated by solving (1) for each value of  $\boldsymbol{\theta}$  and then minimizing (6) over both  $\boldsymbol{\theta}$  and initial conditions  $\mathbf{x}(t_0)$ . Moreover, the GP process provides for robustness against random disturbances of model (1). The methodology was implemented in a Matlab package; see Hooker (2006).

In order to provide gradients for the optimization of  $H$ , we must allow for the fact that the coefficient matrix  $\mathbf{C}(\boldsymbol{\theta})$  is implicitly a function of  $\boldsymbol{\theta}$ . The generalized profiling procedure uses the *implicit function theorem* to define the partial derivatives of  $H$  with respect to components of vector  $\boldsymbol{\theta}$  as follows:

$$\frac{dH}{d\boldsymbol{\theta}} = \left[ \frac{\partial^2 J}{\partial \mathbf{C} \partial \mathbf{C}^T} \right]^{-1} \frac{\partial^2 J}{\partial \mathbf{C} \partial \boldsymbol{\theta}^T}.$$

## 2 Setting up the data for a CollocInfer analysis

The observations themselves are supplied as a matrix with  $n$  rows and  $d$  columns; rows corresponding to times  $t_j$  and columns to observed state variables  $x_i, i \in D_0$ . If there are repeated time series, they can be incorporated as described in Section ??.

Alternatively, the shortcut functions described in Section 7 allow data to be supplied as an array with dimensions  $n, M$  and  $d$  where the middle dimension of size  $M$  refers to replications. For these functions defaults require that a single equation with repeated measurements must be supplied in array format, with the middle dimension being of length 1, otherwise, CollocInfer will misinterpret the input. Note that in using other functions, you will need to concatenate your data observations. Function `sse.setup` will output data in a form that can be used.

It is common for only some of the state variables to be associated with measurements. For example, in the FitzHugh Nagumo (FH) equations, variable  $V$  is voltage and can be measured, but variable  $R$  is a composite of various processes that produce the recovery phase in a neural spike potential, and is therefore not measurable.

When there are unmeasured variables, the data must still be set up as if all variables were measured; that is, as an  $n$  by  $d$  matrix. But those variables which are unmeasured must contain NA's in all locations in the appropriate dimension. For example, in the FH case unmeasured  $R$ , we would include a statement such as `data[,2] = NA`. Alternatively, specific measurement models can be employed. For the case of unmeasured components, or measurements of linear combinations of components the pre-defined `genlin` functions; see Section 4.4.1.

### 3 Setting up the functions: Basic level using squared errors

We now turn to the functions that the user must either program or approximate in order to use the CollocInfer package. If the fit to the data and the fit to the differential equation are evaluated in terms of the sum and the integral of squared errors, respectively, then this task is restricted to setting up the functions or differences for evaluating the functions  $f_i$  in the right side of the differential equation, along with various of its derivatives. This next section explains how to do this.

#### 3.1 Functions for evaluating $f_i(t, \mathbf{x}, \boldsymbol{\theta})$ and its derivatives

A differential equation is defined by the right sides of the differential equations  $f_i(t, \mathbf{x}, \boldsymbol{\theta}), i = 1, \dots, d$ . We will refer to these functions as *right-hand functions* in this manual. In the FitzHugh-Nagumo equations (2), the right-hand functions are  $f_1(t, \mathbf{x}, \boldsymbol{\theta}) = c(V - V^3/3 + R)$  and  $f_2(t, \mathbf{x}, \boldsymbol{\theta}) = (V - a + bR)/c$ , where  $\mathbf{x} = (V, R)'$  and  $\boldsymbol{\theta} = (a, b, c)'$ . (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  $f_i(t, \mathbf{x}, \boldsymbol{\theta})$ . These are obligatory for any application of the CollocInfer package.

In order for the generalized profiling to do its job, the user must also supply functions to evaluate a certain number of partial derivatives of each right-hand function with respect to both the state vector  $\mathbf{x}$  and the parameter vector  $\boldsymbol{\theta}$ , as well as the value of each  $f_i(t, \mathbf{x}, \boldsymbol{\theta})$  itself.

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 elements 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 elements may also be included for purposes specific to an application, but these following named elements 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.

**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.

It is extremely important that you coerce the outputs of each of these functions to be matrices and arrays with the correct number of dimensions and dimension sizes. The R language has a nasty habit of changing the class of matrices and arrays when it indexes them with single indexes, and not telling you. It's easy to get into trouble unless you pay close attention to this. Here's some R code that illustrates the problem:

```
> z = array(0,c(2,2,2))
> class(z)
[1] "array"
> class(z[1,])
[1] "matrix"
> class(z[1,1])
[1] "numeric"
```

The four arguments to each function are

**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 state values corresponding to the times argument. The matrix has  $d$  columns corresponding to state variables and  $n$  rows corresponding to times. Note that this argument contains state values, not data values, and therefore all  $d$  columns should contain only numeric data.

**p:** a vector 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 usually be a list object.

But, in addition, you may wish to use the names list object to supply a right hand side evaluation function for function `lsoda`. We use this function to approximate the solution to a differential equation given initial values for the states of the system, and this we often do after parameters defining the system have been estimated and we want to display what a solution to our estimated equation looks like. Function `lsoda`, however, does not normally use the `more` argument, and so you may wish to also provide a version of function `fn`, which might be called `fn.ode`, which is identical to `fn` except for not using the final argument. For an illustration of this, display the function `make.fn()` that sets up the FitzHugh-Nagumo equation list object.

We illustrate the setup of these functions by the simplest of differential equation systems, a single first order constant coefficient system:

$$Dx(t) = -\beta x(t)$$

Note how we coerce the returned value of each function to a matrix or array of the appropriate dimensions.

```

make.01fn <- function()
{
# set up functions for Dx = -beta*x

01fun = function(times, x, p, more) {
  n = length(times); d=1; npar=1
  beta = p(1)
  f = matrix(-beta*x,n,d)
  return(f) }

01dfdx = function(times, x, p, more) {
  n = length(times); d=1; npar=1
  beta = p(1)
  dfdx = array(-rep(beta,n),c(n,d,d))
  return(dfdx) }

01dfdp = function(times, x, p, more) {
  n = length(times); d=1; npar=1
  dfdp = array(-x,c(n,d,npar))
  dfdp[,1,] = cbind(dfdp1,dfdp2,dfdp3)
  return(dfdp) }

01d2fdx2 = function(times, x, p, more) {
  n = length(times); d=1; npar=1
  d2fdx2 = array(0,c(nobs,d,d,d))
  return(d2fdx2) }

01d2fdxdp = function(times, x, p, more) {
  n = length(times); d=1; npar=1
  d2fdxdp = array(-1,c(n,d,d,npar))
  return(d2fdxdp) }
return(list(fn = 01fun, dfdx = 01dfdx,
           dfdp = 01dfdp, d2fdx2 = 01d2fdx2,
           d2fdxdp = 01d2fdxdp))
}

```

It is wise to include argument checks in at least 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.

Here is a function that evaluates the two right-hand functions for the FitzHugh-Nagumo system with checks. This system is a bit special in that the `times` argument has no role to play and the function does not need whatever is in the `more` argument.

```

fhn.fun <- function(times, x, p, more) {
# check arguments for various conditions

```

```

if (dim(x)[1] != length(times))
  stop("Length of times not equal to number of rows of x.")
if (dim(x)[2] != 2) stop("Argument x does not have 2 columns.")
if (any(is.na(x))) stop("Argument x contains NA or NaN values.")
if (length(p) != 3) stop("Argument p is not of length 3.")
# compute right hand function values in matrix Dx
Dx = x # The output is a matrix with the same dims as x
Dx[,1] = p[1]*(x[,1] - x[,1]^3/3 + x[,2]) # V derivative
Dx[,2] = -(x[,1] - p[1] + p[2]*x[,2])/p[3] # R derivative
return(Dx)
}

```

The return statement for function `make.fhn` in the `CollocInfer` package that defines the right-hand functions is

```

return(list(fn      = fhn.fun,
           dfdx     = fhn.dfdx,
           dfdp     = fhn.dfdp,
           d2fdx2   = fhn.d2fdx2,
           d2fdxdp  = fhn.d2fdxdp))

```

In each argument of function `list`, the string before the `=` must be exactly as shown, but the string after begins the user-defined name of the list containing the required functions followed by `.` and then the member name within that list.

The `CollocInfer` package requires the least effort if the fit to the data in the first term is a weighted sum of squared residuals and the fit to the state derivatives in its second term is defined by the integral of the squares of the errors or residuals, that is,

$$J(C|\boldsymbol{\theta}) = \sum_{i \in D_o} \sum_{j=1}^n w_{ij} [y_{ij} - x_i(t_j)]^2 + \sum_{i=1}^d \lambda_i \int \left[ \frac{d}{dt} x_i - f_i(t, \mathbf{x}(t), \boldsymbol{\theta}) \right]^2 dt.$$

Probably most applications will use this squared error criterion. Setting up the right-hand function evaluators is pretty much all the coding that is required of the user, and at this point the reader could proceed to subsequent sections of this manual, or even directly to the worked examples in Sections 9 and 11.

### 3.2 Computing derivatives by differencing: `make.findif.loglik`

When the equations are as simple as they are for the FitzHugh-Nagumo system, many of us can get the derivatives right without too much trouble, but if the systems are large or the equations complex, the calculus can become formidable. Even if one thinks one has them right, it can be a comfort to be able to check the results without too much effort.

`CollocInfer` function `make.findif.loglik` creates a list object whose members calculate finite difference approximations to the various required derivatives of the error sum of squares fit measure in the first term of  $J(C|\boldsymbol{\theta})$ . Finite differences are

defined by adding a small amount  $\epsilon$  to the quantity with respect to which the derivative is to be taken. For example, the derivative of  $f$  with respect to  $x$  is approximated by

$$\frac{\partial f}{\partial x} \approx \frac{f(t, x + \epsilon, \boldsymbol{\theta}) - f(t, x, \boldsymbol{\theta})}{\epsilon}$$

The user must supply the actual value of  $\epsilon$  to be used.

Since all quantities are perturbed by the same constant, it is essential that all quantities to be differenced are on roughly the same scale. For the FitzHugh-Nagumo system, this is  $V$ ,  $R$ ,  $a$ ,  $b$  and  $c$ , and in fact they are on about the same scale, namely one or unity. If this requirement is not satisfied, it is usually possible to revise the equations and parameters so that it is.

If  $d$  and  $p$  are not overly large, this is not computationally intensive. Of course, accuracy suffers, but for most purposes accuracy to within about four significant digits is all that is required. If all the variables to be differenced are on the scale of one or unity, then  $\epsilon = 0.0001$  might be about right. Some experimentation to see that the required accuracy is achieved may be achieved by comparing results for one or two easily calculated derivatives.

Function `make.findif.loglik()` produces a `lik` object with members having names `fn`, `dfdx`, `dfdy`, `dfdp`, `d2fdx2`, `d2fdxdy`, `d2fdy2`, `d2fdxdp`, `d2fdydp` to calculate the appropriate derivatives by fixed-step finite differencing. These functions require the named list object `more` to contain entries

**fn:** a function that computes the values of the right-hand functions.

**eps:** the change  $\delta_\theta$  in the parameter to be used to compute the finite differences.

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

### 3.3 Setting up a first FitzHugh Nagumo example

Here we give an example involving only minimal setup and use of `CollocInfer` functions. We use functions `create.bspline.basis` and `smooth.basis` from the R package `fda` to set up the analysis.

We assume that both variables are measured at 41 times spaced apart by  $1/2$  of a time unit. Here we set up the observation times in vector `times`, and define the state variable names.

```
times = seq(0,20,0.5)
FHN.xnames = c('V', 'R')
```

The data array `FHN.data` must have 41 rows and two columns. Its rows correspond to 41 times of observation and columns to the variables,  $V$  and  $R$ , respectively. Section 9 shows how data for this system can be simulated.

Next we invoke the pre-coded function `make.fhn()`, which is distributed with the `CollocInfer` package in the `demo` folder. This function generates the named list object `FHN.fn` that has as its members the functions to evaluate the right-hand functions defined in (2).

```
FHN.fn = make.fhn()
```

It might be helpful to look at this function by typing `make.fhn()` in the R command window to see how the `FHN.fn` is constructed, so that you can see how you might construct your own objects. What would happen, for example, if you deleted the  $V^3$  term in the first equation? Or changed the cube function to something else, such as `exp(V) - 1`?

The collocation process requires a basis system  $\phi$  for representing the state functions. This code defines an order 3 B-spline basis function with a knot at every fourth observation time.

```
FHN.knots = seq(0,20,2)
FHN.order = 3
FHN.nbasis = length(knots) + FHN.norder - 2
FHN.range = c(0,20)
FHN.basis = create.bspline.basis(FHN.range, FHN.nbasis,
                                FHN.order, FHN.knots)
```

Initial values for the coefficients in matrix  $\mathbf{C}$  define state vector  $\mathbf{x} = \phi\mathbf{C}$  are obtained by smoothing the raw data using the R package `fda` function `smooth.basis` and then extracting the coefficients from the functional data object.

```
FHN.fdnames = list(NULL, NULL, FHN.xnames)
FHN.xfd = smooth.basis(times, FHN.data, FHN.basis,
                      fdnames=FHN.fdnames)$fd
FHN.coefs = FHN.xfd$coefs
```

We also need to supply some initial values for the parameters, supplied here in vector `FHN.pars0`. The second statement attaches labels to the three parameters.

```
FHN.pars0 = c(0.2, 0.2, 3.0)
names(FHN.pars0) = c('a', 'b', 'c')
```

Profiled estimation is now completed by a call to function `Profile.LS`. Smoothing parameter values  $\lambda$  are set to 10000 for both variable. The function returns a named list object `resultList` containing, among other things, the final parameter estimates  $\hat{\theta}$ , the coefficient matrix estimate  $\hat{\mathbf{C}}$ , and the residual values  $r_{ij} = y_{ij} - x_i(t_j)$ .

```
FHN.lambda = 1e4*c(1,1)
resultList = Profile.LS(FHN.fn, data, time, pars0, FHN.coefs,
                      FHN.basis, FHN.lambda)
```

Finally, we extract from this list the results that we need for plots, further analyses, and so on.

```
FHN.pars = resultList$pars
FHN.coefs = resultList$coefs
FHN.residuals = resultList$residuals
```

We can avoid all the derivative coding in the definition of `FHN.fn`. The member of `FHN.fn` that computes the state variables is `FHN.fn$fn`, and we could replace the first argument of `Profile.LS` by `FHN.fn$fn` to use differencing to approximate these derivatives.

## 4 Setting up functions for customized fit measures

### 4.1 User-defined fit measures

In this `CollocInfer` R package, we make use of the same framework, but extend the Matlab version in several useful ways.

The GP procedure described above uses a least squares measures of fit. In this package we permit the user to employ other measures to define the quality of fit for both terms in (5). In the lower or inner optimization step, where  $\boldsymbol{\theta}$  is fixed,  $\mathbf{C}$  minimizes

$$J(\mathbf{C}|\boldsymbol{\theta}) = \sum_{i \in D_o} \sum_{j=1}^n w_{ij} F\{y_{ij}; g[\Phi(t_j)\mathbf{c}_i]\} + \sum_{i=1}^d \lambda_i \int P \left[ \frac{d}{dt} \mathbf{c}_i \Phi(t), f_i(t, \Phi(t)\mathbf{C}, \boldsymbol{\theta}) \right] dt. \quad (7)$$

This formulation introduces three new functions, each which can be defined by the user.

1. Fit function  $F$  is a measure of fit to the data. It will often be the sum of negative log density function values

$$F(\mathbf{y}_j) = - \sum_{i \in D_o} \log p_i(y_i|x_i(t_j), \boldsymbol{\theta}).$$

In this case,  $F(\mathbf{y}_j)$  is the *negative log likelihood* of the observations associated with the  $j$ th time value, and the sum over  $j$  is the *total negative log likelihood* of the data. Note that we only sum over the state variables that are observed.

2. The function  $P$  in the second term quantifies failure to fit the differential equation, and can also be thought of as representing a negative log likelihood for  $\mathbf{x}$ . However, in order to avoid any possible confusion about which likelihood we might be referring to, we shall refer to  $P(t)$  as measuring the *roughness* or *regularity* of  $\mathbf{x}$  at  $t$ , as is usual in functional data analysis, nonparametric regression and most other branches of statistics. More general roughness measures using higher order derivatives or spatial co-ordinates are possible in `CollocInfer`.
3. The function  $g$  allows the possibility that the differential equation is actually a model for a transformation of the process generating the data. For example,  $g(x) = \exp(x)$  indicates that the state value  $x(t)$  is actually a model for log of the function underlying the observed data at time  $t$ , and allows the user to ensure that the fit to the data will be *positive*. The exponential and other commonly used transformations are pre-specified in `CollocInfer`, as is the *identity transformation*  $g(x) = x$ , which is the default choice. However, `CollocInfer` also permits the user to define  $g$  for unforeseen circumstances.

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 differencing at all.

Suppose, now, that the user wants to define customized measures of fit, indicated in (7), by defining at least one of  $F$  and  $P$ , and possibly  $g$ .

We refer to the data fit measure  $F$  in the first term as the `lik` fit, the derivative fit measure  $P$  in the second term as the `proc` fit and the function  $g$  as the *link function*. In particular, the `lik` fit will often be defined as  $-\log p_y$ , the negative of the sum of the logarithms of the probability density function values that model the process conjectured to generate the residuals. Some applications will, of course, require customized choices for only one `lik`, `proc`, and  $g$ , with the other being left to be the default definition.

## 4.2 Defining `lik` objects to assess data fits

A customized `lik` fit measure  $F$  requires the coding of a set of functions for evaluating its values and those of various of its 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  $f_i(t, \mathbf{x}, \boldsymbol{\theta})$  described above in Section 3.1. And they will also be used in the `proc` list object described below in Section 4.3, as well in other named list objects.

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

**bvals**: an object defining the required basis values; this may vary depending on **fn**.

**fn**: a function that calculates the data fit measure  $F$  at time  $t_j$  taken over the observed state variables, such as the negative log likelihood of the residuals. This function returns a scalar.

**dfdx**: a function that calculates the partial derivatives of **fn** with respect to the values of the state variables  $\mathbf{x}$ . It returns a vector of length  $d$ .

**dfdp**: a function that calculates partial derivatives with respect to parameters. It returns a vector of length  $p$ .

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

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

**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 manual.

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  $\mathbf{y}$ :

**dfdy**: a function that calculates the partial derivatives of **fn** with respect to data values. It returns a vector of length  $d$ , 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 a matrix of size  $d \times d$ , 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 state values. It returns a matrix of size  $d \times d$ , but entries corresponding to unobserved variables are not used.

The arguments for the evaluator functions **fn** to **d2fdydp** are the same as those for the right-hand function evaluators  $f_i(t, \mathbf{x}, \boldsymbol{\theta})$  described in Section 3.1 above, but augmented by a first argument specifying the data and by a matrix of basis function values. That is, they are

**y**: an  $n \times d$  matrix or an  $n \times N \times d$  array of observation values

**times**: a vector of length  $n$  containing times of observations

**x**: an  $n \times d$  matrix of state values corresponding to the times argument.

**p**: a vector of length  $p$  containing parameter values

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

**bvals**: an  $n \times K$  matrix containing the values of the basis functions at the observation times

These functions are returned in a named list object. The names for the elements or entries in the list are the same as the list returned for the right-hand functions, described in Section 3.1.

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 slots. For each of these, the slot **more** is required to have specific entries that are detailed below.

As in the basic function setup, we can here use CollocInfer function **make.findif.loglik** to either check the derivative calculations, or even to substitute for programming them entirely.

#### 4.2.1 SSElik: The ordinary least squares lik object

This is not so much a likelihood as straight squared error. Function **make.SSElik()** creates a list with entries **fn**, **dfdx**, **dfdy**, **dfdp**, **d2fdx2**, **d2fdxdy**, **d2fdy2**, **d2fdxdp**,

`d2fdydp`. These functions calculate

$$l(\mathbf{y}, t, \mathbf{x}, p) = \sum_{i \in D_o}^d w_i (y_i - f_i(t, \mathbf{x}, p))^2$$

and its corresponding derivatives. They require `more` to contain functions `fn`, `dfdx`, `dfdp`, `d2fdx2`, `df2dxdp` and `more$more` for further arguments. 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. Function `fn` is vector valued and returns a  $n \times d$  array. 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 an element `weights` which contains a matrix of the  $w_i$  which should be of the same dimension as  $Y$ . It 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.

#### 4.2.2 multinorm: Generalized least squares data fits

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

$$\ell(\mathbf{y}, t, \mathbf{x}, \boldsymbol{\theta}) = \{[\mathbf{y} - \mathbf{f}(t, \mathbf{x}, \boldsymbol{\theta})]^T \mathbf{V}^{-1}(t, \mathbf{x}, \boldsymbol{\theta})[\mathbf{y} - \mathbf{f}(t, \mathbf{x}, \boldsymbol{\theta})]/2 - \log |\mathbf{V}(t, \mathbf{x}, \boldsymbol{\theta})|\}/2.$$

This is a generalization of the `SSElik` functions to correlated processes who's 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 member names `fn`, `dfdx`, `dfdy`, `dfdp`, `d2fdx2`, `d2fdxdy`, `d2fdy2`, `d2fdxdp`, `d2fdydp` which calculate a multivariate normal distribution.

These functions require `more` to contain `fn`, `dfdx`, `dfdp`, `d2fdx2`, `df2dxdp` 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  $\mathbf{V}(t, x, \boldsymbol{\theta})$ . Since  $\mathbf{V}(t, x, \boldsymbol{\theta})$  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  $\mathbf{V}$ . 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.

But on occasion the distribution of the data or requirements for the roughness penalty will suggest other choices of fit measures that the user can define. In this case, evaluator functions must also be set up for these user-defined fit measures. The next subsection shows how to do this, but `CollocInfer` nevertheless has built-in procedures for doing this for the least squared error case, and these can be viewed by typing `make.SSElik()` and `make.SSEproc()`. Looking at these can be a useful way to see how to set up one's own customized versions.

### 4.3 Defining proc objects for assess equation fits

The `proc` object stores functions to calculate the second term, the roughness penalty  $P$ . In fact, this task is not so different from what is required to compute the data-fit `lik` term. An examination of both the least squares version of the inner optimization criterion  $J$  in (5) and its more general version in (7) indicates that we

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

#### 4.3.1 Quadrature points $t_q$ and weights $w_q$ for numerical integration

Actually, 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

$$\int \left[ \frac{d}{dt} \Phi(t) \mathbf{c}_i - f_i(t, \Phi(t) \mathbf{C}, \boldsymbol{\theta}) \right]^2 dt \approx \sum_q w_q \left[ \frac{d}{dt} \Phi(t_q) \mathbf{c}_i, f_i(t, \Phi(t_q) \mathbf{C}, \boldsymbol{\theta}) \right]^2 \quad (8)$$

in the least squares case (5) and

$$\int P \left[ \frac{d}{dt} \mathbf{c}_i \Phi(t), f_i(t, \Phi(t) \mathbf{C}, \boldsymbol{\theta}) \right] dt \approx \sum_q w_q P \left[ \frac{d}{dt} \Phi(t_q) \mathbf{c}_i, f_i(t, \Phi(t_q) \mathbf{C}, \boldsymbol{\theta}) \right] \quad (9)$$

in the more general setting (7).

Numerical quadrature plays an absolutely essential role in the collocation approach, or indeed in any method of approximating a solution to a differential equation. The user must supply 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 knowledge of the topic of numerical quadrature.

However, where 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 and naive approach to quadrature is called the *trapezoidal rule*. Other quadrature methods such as Simpson's Rule or Gaussian quadrature are more accurate but more complicated to set up.

#### 4.3.2 Defining the proc functions and their arguments

As in the definition of the `lik` object, the `proc` is a named list, some of the names being specifically required by the `CollocInfer` package, and of these the majority being user-defined functions.

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

- `fn`: a function that calculates the log probability of the process; returns a scalar.
- `dfdc`: a function that calculates the derivatives of `fn` with respect to coefficients in `C`, returns a vector of length  $dK$ .
- `dfdp`: a function that calculates the derivatives with respect to parameters in  $\theta$ , 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  $dK \times dK$
- `d2fcdp`: 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 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` higher derivatives of the basis functions at the quadrature points.

All of the above functions take the following arguments

- `coefs`: the current estimate of the coefficients
- `bvals`: as given in the `bvals` slot 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
  - `qppts`: a vector of length  $Q$  containing quadrature points  $t_q$  where the penalty is to be evaluated.
  - `weights`: a vector of length  $q$  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.

CollocInfer functions `make.findif.lik` and `make.findif.proc` is provided to either check the derivative calculations in the `proc` object using finite differencing, or even to substitute for programming these derivatives entirely. The use of this function is nearly identical to that for function `make.findif.loglik`, described in Section 3.2.

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.

#### 4.3.3 SSEproc: The ordinary least squares proc object

This is the analogue of the `SSElik`, based on approximation to the integrated squared error version of the roughness penalty

$$P = \sum_{i=1}^d \sum_{q=1}^Q w_q \left[ \frac{d\Phi}{dt}(t_q) \mathbf{c}_i - f_i(t, \Phi(t_q) \mathbf{C}, \boldsymbol{\theta}) \right]^2.$$

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  $t_j$  where the penalty is to be evaluated.

**weights:** a matrix giving the quadrature weights  $q_{ij}$

#### 4.3.4 Cproc and Dproc: generalized least squares proc objects

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

$$P_i = \sum_{q=1}^Q w_q \ell \left[ \frac{d\Phi}{dt}(t_q) \mathbf{c}_i; f_i(t, \Phi(t_q) \mathbf{C}, \boldsymbol{\theta}) \right]$$

It can be used to take any of the 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

```
proc = make.Cproc()
proc$more = make.SSElik()
```

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

The `Dproc` functions provide similar functionality to `Cproc` but for discrete-time processes with values at  $L$  equally-spaced time points. These calculate

$$P_i = \sum_{j=1}^{L-1} l[\Phi(t_{j+1})\mathbf{c}_i; f_i(t, \Phi(t_i)\mathbf{C}, \boldsymbol{\theta})].$$

We often define  $\Phi$  to be a sequence of constant functions with breaks in the mid-points between the  $t_j$ . This is then equivalent to estimating the discrete state of the system.

The arguments for function `Dproc` are mildly different:

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

**more\$qpts** is an  $L - 1$  vector of times.

The `proc` named list defined by function `Dproc` also requires the same members `fn`, `dfdx`, `dfdy`, `dfdp`, `d2fdx2`, `d2fdxdy`, `d2fdy2`, `d2fdxdp`, `d2fdydp`, and `more` as are 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

```
bvals$bvals = basisvals[1:(nrow(basisvals)-1),]
bvals$dbvals = basisvals[2:nrow(basisvals),]
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.

#### 4.4 Link functions $g$ for indirect data–model relations

It happens often that the data we collect are only indirectly related to the process that we define by the differential equation.

For example, experiments in the physical sciences often involve the measurement of **magnitudes** such as mass, density, heat, work and so on that have a meaningful zero and are otherwise intrinsically positive. But linear differential equations, the

easiest ones to work with, cannot be prevented from having negative values  $x(t)$  in their solutions. An option in this case is to fit the data with a transformation of the value of the differential equation, and in the nonnegative case, we would logically use  $g(x) = \exp(x)$  for this purpose. Likewise, a chemist might measure the concentration of a chemical species in the output of a chemical reactor, and record these concentrations as percentages. In this case, an effective transformation would be  $g(x) = 100 \exp(x)/[1 + \exp(x)]$ .

Another example arises when the data fit provided by state  $x$  is augmented by a contribution from a covariate  $z$ , so that  $g(x) = x + \beta z$  where  $\beta$  is a regression coefficient that must be estimated from the data. In this event, parameter  $\beta$  is included within the parameter vector  $\theta$ , and the link function therefore is dependent on the parameter vector as well as on the value of  $x$ . This, too, can be provided for in a user-specified link function.

On the other hand, doing nothing at all to  $x$  is also an option, and the transformation  $g(x) = x$  is called the *identity transformation*.

Each link function must also be able to compute, in addition to  $g(x)$ , the values of various partial derivatives involving  $\theta$  and  $y$ , since the fitted value of  $x$  also depends on these quantities. CollocInfer specifically requires the user to provide  $dg/dx$ ,  $d^2g/dx^2$ ,  $\partial g/\partial\theta$ ,  $\partial g/\partial y$  and  $\partial^2 g/\partial x \partial\theta$ .

The link function is passed to various functions through the `more` named list in the `lik` object as a named list with members `fn`, `dfdx`, `dfdp`, `d2fdx2`, `d2fdxdp`. An example of how this link named list is defined for the exponential transformation can be found in the CollocInfer function `make.exp`, and function `make.id` sets up the identity transformation.

At the time of writing of this manual, CollocInfer has predefined functions `make.id` and `make.exp` for only the identity and exponential transformations, respectively.

An example of the use of the identity and exponential transformations can be found in the CollocInfer function `SSEsetup`, which sets up `lik` and `proc` objects for the error sum and integral of squared residuals case, respectively. These three statements set up the `lik` object and use an argument `pos` in the call to `SSEsetup` to allow the user to switch between the identity (the default) and the exponential transformation.

```
lik = make.SSElik()
if (pos==0) lik$more = make.id()
else      lik$more = make.exp()
```

Another example of the use of link functions is in the `multinorm.setup` function.

CollocInfer function `make.findif.ode` provides finite-difference estimation of the derivatives when they are too cumbersome, or computationally expensive, to evaluate analytically.

CollocInfer function `make.findif.ode()` creates a list with slots `fn`, `dfdx`, `dfdp`, `d2fdx2`, `d2fdxdp` that calculate derivatives by naive finite differencing. The element `more` should contain

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

**eps:** the time interval  $\delta t$  to be used in the differencing.

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

#### 4.4.1 genlin

These functions calculate

$$\mathbf{f}(t, \mathbf{x}, p) = A(p)\mathbf{x}(t) + B(p)\mathbf{u}(t)$$

a generic linear combination of the states, plus a linear combination of  $d_e$  known external inputs  $\mathbf{u}$ . This is useful, for example, when the measurement is of some proportion of a state or for linear dynamics.

`make.genlin()` creates a list with slots `fn`, `dfdx`, `dfdp`, `d2fdx2`, `d2fdxdp`. In addition, the element `more` can specify a flexible structure for  $A$ . `more` is a list with slots

**mat** a  $d \times d$  matrix for  $A(0)$  with constant entries not affected by the parameters. Defaults to zero if not specified.

**sub:** a  $p' \times 3$  array giving in order the row position, column position and index of parameters to be used in  $A$ ; a row with elements  $(i, j, k)$  specifies that  $A_{ij} = p_k$ .

Defaults to filling in all entries of  $A$  in row-order starting with the first element of  $p$ .

**force:** a list of length  $d_e$  containing either functional data objects, or functions to evaluate  $\mathbf{u}(t)$ . Defaults to NULL.

**force.mat:** a matrix for  $B(0)$ ; defaults to identity if `force` is not NULL.

**force.sub:** as in `sub` for  $B(p)$ . Defaults are

- if NULL continue to fill in  $B(p)$  in row order, starting from  $d^2 + 1$ .
- if a vector, the  $i$ th position having value  $j$  specifies  $B_{ij} = p_{d+i}$ .
- if a matrix, the  $i$ th row being  $(j, k)$  specifies  $B_{ij} = p_k$ .

## 4.5 Variance functions for defining generalized least squares fits

The `multinorm` functions also require a variance function to be calculated. Sometimes, the variance may itself be state-dependent (see, for example, the SEIR equations below), but it will also frequently be treated as constant

$$V(t, \mathbf{x}, p) = V(p).$$

`CollocInfer` function `make.cvar()` creates a list with slots `var.fn`, `var.dfdx`, `var.dfdp`, `var.d2fdx2`, `var.d2fdxdp` defining constant variance parameters. These require `more` to be a list containing

**mat:** the matrix for  $V(0)$ . Defaults to zero if `more$sub` is defined, or identity otherwise

**sub:** as in the matrices for  $A$  and  $B$  in `genlin`. This should be a  $p \times 3$  array. A row containing  $(i, j, k)$  implies  $V_{ij} = V_{ji} = p_k$ .

CollocInfer function `make.findif.var` provides finite difference estimation for the variance function when it is too expensive or intractable to evaluate analytically.

CollocInfer function `make.findif.var()` creates a list with elements `fn`, `dfdx`, `dfdp`, `d2fdx2`, `d2fdxdp`. The element `more` should contain

**fn:** the function to be differenced.

**eps:** the time step for finite differencing.

**more:** any further objects to be passed to `fn`.

## 5 Confidence intervals for estimates of parameters in $\theta$

Ramsay et al. (2007) suggests confidence intervals based on standard non-linear least squares methods. The approximate covariance matrix for the parameter estimates may be given by

$$\text{Cov}(\hat{\theta}) \approx \hat{\sigma} \left[ \mathbf{J}^T \mathbf{J} \right]^{-1} \quad (10)$$

where  $\mathbf{J}$  is an  $n \times p$  matrix containing the partial derivatives of the predicted values with respect to the parameters. In the case of the methods described above, this amounts to

$$\mathbf{J} = \Phi(\mathbf{t}) \frac{d}{d\theta} \mathbf{C}(\theta).$$

But estimate (10) may be too optimistic for two reasons:

1. For stochastic systems, the estimate accounts only for the variance resulting from observational noise; it does not account for process noise.
2. Covariance estimate (10) is based upon the assumption that the rows of  $\mathbf{J}$  are independently sampled at the true parameters. For  $\lambda$  small, this is not tenable, since the estimates for  $C(\theta)$  uses all the observations.

Instead, we combine Newey-West and sandwich estimators to provide approximate covariance matrices. Modifying the usual MLE asymptotics, we write

$$(\hat{\theta} - \theta) = \mathbf{V}^{-1} \mathbf{J} \mathbf{1}.$$

Here  $\mathbf{1}$  is a column of 1's providing a summation operation.  $\mathbf{V}$  is the second derivative of the expected objective function at  $\theta$  and we take  $\mathbf{J}$  to be the derivative of the objective function. Note that in contrast to  $\mathbf{J}$  above,  $\mathbf{J}$  may combine multiple measurements taken at the same time points.

In the squared-error case, we can obtain a consistent estimate for  $\mathbf{V}$  from

$$\mathbf{V} \approx \mathbf{J}^T \mathbf{J}.$$

Assuming a central limit theorem for  $\mathbf{J}$  we can obtain a covariance estimate for  $\hat{\theta}$  from the sandwich estimator:

$$\text{Cov}(\hat{\theta}) \approx \mathbf{V}^{-1} \text{Cov}(\mathbf{J} \mathbf{1}) \mathbf{V}^{-1}.$$

If the rows of  $\mathbf{J}$  are independent the covariance on the right hand side is  $\mathbf{V}$ , yielding the usual inverse Hessian. When this is not the case, we can instead employ a Newey-West estimate (Newey and West, 1987):

$$\text{Cov}(\mathbf{J} \mathbf{1}) \approx \hat{\Omega}_0 + \sum_{k=1}^m \left( 1 - \frac{k}{m+1} \right) \left( \hat{\Omega}_k + \hat{\Omega}_k^T \right), \quad \hat{\Omega}_k = \frac{1}{n} \sum_{t=k+1}^n \mathbf{J}_{t-k} \mathbf{J}_t^T$$

Under mild mixing assumptions, this estimate is consistent when  $m = o_p(n^{1/4})$ .

This estimate effectively included successively down-weighted estimates of the covariance at distance  $k$  between the rows of  $\mathbf{J}$ . The particular formulation ensures the positive semi-definiteness of the estimator. The implication that the rows of  $\mathbf{J}$  are only locally covariant appears reasonable; dependence among the rows stems from the estimation of  $C(\boldsymbol{\theta})$  (at a given  $\boldsymbol{\theta}$ ) which will themselves be only locally dependent.

However, it should be noted that this approximation is based on the score being a sum of (marginally) identically distributed variables. This is reasonable only when the observations are taken at equi-spaced intervals and the same components are always measured. When the measurements are not equi-spaced, or different components are measured at different times, similar approximations may be made, but these will be complicated and must be tailored to the specific situation.

## 6 Optimizing functions and the functions they call

The generalized profiling method works by optimizing two sets of parameters: the coefficients  $\mathbf{c}_i, i = 1, \dots, d$  defining the estimated state variables  $x_i$ , and the parameters contained in the parameter vector  $\boldsymbol{\theta}$ . This optimization proceeds in two steps.

In the inner step, the coefficients are estimated holding  $\boldsymbol{\theta}$  fixed by optimizing criterion  $J$  defined in (5) and (7). The main function for doing this in CollocInfer is function `inneropt`.

In the outer step, criterion  $H$  defined in (6) is optimized with respect to the parameters in  $\boldsymbol{\theta}$ , taking account of the fact that the coefficients in  $\mathbf{C}(\boldsymbol{\theta})$  are implicitly functions of  $\boldsymbol{\theta}$ . The main function for doing this in CollocInfer is function `outeropt`.

R has many functions and packages of functions dedicated to the numerical optimization of functions, and the CollocInfer packages permits the user to choose from a wide number of these within both function `inneropt` and function `outeropt`. Why?

The collocation process can generate challenging optimization problems for two reasons. First, there are typically a large number of coefficients in  $\mathbf{C}$ , so that the inner optimization can be a high-dimensional, and if poor starting values are supplied, the optimization may either fail or taken an unacceptable length of time to find an optimum, depending in part on the strategy used by the optimizing function. Some methods work better than other for high dimensional problems, and the user may find it necessary to experiment with a few before deciding which to use.

The R optimization methods made available are

- `nlminb`
- `optim`
- `maxNR`
- `trust`
- `SplineEst.NewtRaph` (a method developed specially for CollocInfer)

Each estimation problem is unique, and some of these functions are still being improved by their original designers, so that we do not wish to indicate any preference among these, although Cao and Ramsay (2009) do suggest that the performance of `nlminb` was decidedly inferior to that of `optim` in study of the linear mixed effects model, which has some similarities to the collocation problem. Further details and information about each optimization function can be obtained by consulting the help pages for the function, such as `help("optim")`.

In the outer optimization of  $H$ , the number parameters in  $\boldsymbol{\theta}$  is typically much smaller, but the fact that the value of  $H$  depends both explicitly and implicitly on  $\boldsymbol{\theta}$  (through  $\mathbf{C}(\boldsymbol{\theta})$ ) can mean that the surface defined by  $H$  can have complex topographical features, including multiple local optima, that can make even low-dimensional optimization challenging. Again, the possibility of choosing among

several optimizing strategies can be important, and is for that reasons supplied to the user in the two functions.

CollocInfer allows for the possibility that only a subset of parameters in  $\theta$  are to be optimized, with the remainder being held fixed.

These two factors imply that beginning the optimization with good initial values for both  $\mathbf{C}$  and  $\theta$  can make the difference between success and failure. Consequently, CollocInfer also offers a number of functions dedicated to the preliminary estimation of starting values, especially for  $\mathbf{C}$ , from the data.

Both functions `inneropt` and `outeropt` actually do little except select among optimization functions and set up the results of the optimization for output. However, each optimization function has its own peculiarities, and therefore a variety of lower-level functions are provided by CollocInfer to be called by various optimizing functions, and will be described below after we have indicated the arguments that can be provided to `inneropt` and `outeropt`.

## 6.1 Inner optimization of $J$ to estimate coefficients: `inneropt`

Function `inneropt` optimizes the inner optimization criterion  $J$  defined in (5) by selecting among a number of optimization functions available in R.

The arguments are

**data:** A matrix (unreplicated data) or array (replicated data) of observed data values.

**times:** A vector of  $n$  observation times for the data.

**pars:** A vector of  $p$  current values of the parameters in  $\theta$  to be estimated.

**coefs:** A matrix or array containing the initial estimates of the coefficients in  $\mathbf{C}$ .

**lik:** the `lik` object (a named list) defining the observation process.

**proc:** the `proc` object (a named list) defining the state process.

**in.meth:** a string designating the inner optimization function: currently one of ‘nlminb’, ‘maxNR’, ‘optim’ (uses the ‘BFGS’ option), ‘trust’ or ‘SplineEst’.

The last calls `SplineEst.NewtRaph`. This is fast but has poor convergence.

**control.in:** A control object that controls the inner optimization function.

Argument `control.in` should contain whatever control parameters are appropriate for the optimization routine being called. When these are given as separate arguments in the optimization function call (as in `maxNR`), they should be listed in `control.in` and are then unpacked. Leaving `control.in=NULL` results in default values which are not always effective. If `in.meth=NULL` it defaults to ‘nlminb’.

Function `inneropt` returns a named list with members

**coefs:** The matrix  $\mathbf{C}$  containing the coefficients of the fit for all states.

**res:** The result of the optimization, typically a list object. This is specific to the optimization routine used, and the help page for the optimization function must be consulted for details on what it contains.

Depending on which optimization routine is invoked, the following functions are called by the optimizing function:

**SplineCoefsErr:** Computes the estimated state function values  $\mathbf{x}(t)$ .

**SplineCoefsDC:** Computes the derivative of  $\mathbf{x}(t)$  with respect to the coefficients of the basis expansion.

**SplineCoefsDP:** Computes the derivative of **SplineCoefsErr** with respect to the parameters in the system. May be used as an alternative to **ParsMatchErr** and **ParsMatchDP**.

**SplineCoefsDC2:** Computes the Hessian of **SplineCoefsErr** with respect to the coefficients of the basis expansion.

**SplineMaxLik:** Computes function returning the output of **SplineCoefsErr** with the output of **SplineCoefsDC** and **SplineCoefsDC2** as a named list with member names **gradient** and **hessian**, respectively.

Each of these lower level functions takes arguments

**coefs:** the current value of the coefficients as a vector of length  $dK$  (coefficient nested within variables) or a  $K \times d$  matrix

**times:** the observation times given as a  $n$ -vector

**data:** the observed data given as an  $n \times d$  matrix (unreplicated) or as an  $n \times N \times d$  array (replicated)

**lik:** the **lik** named list object for the data fitting term

**proc:** the **proc** named list object for the roughness penalty term

**pars:** a vector containing the  $p$  parameters in  $\theta$

**sgn:** a scalar variable taking values +1 or -1 indicating if the objective function should be maximized (-1) or minimized (+1). Defaults to +1.

## 6.2 Outer optimization $H$ to estimate parameters: **outeropt**

Function **outeropt** optimizes criterion  $H$  defined in (6) with respect to the parameters in  $\theta$ ; and, like function **inneropt**, it essentially selects among R optimization functions. In the special case of squared error models (5), nonlinear least squares optimization can be more efficient and more convenient. Functions **Profile.LS** uses this strategy, along with function **Profile.GausNewt**, and these are described in Section 6.4 below.

Function **outeropt** is called with these arguments:

**data:** A matrix (unreplicated data) or array (replicated data) of observed data values.

**times:** A vector of  $n$  observation times for the data.

**pars:** A vector of  $p$  current values of the parameters in  $\theta$  to be estimated.

**coefs:** A matrix or array containing the initial estimates of the coefficients in  $\mathbf{C}$ .

**lik:** the `lik` object (a named list) defining the observation process.

**proc:** the `proc` object (a named list) defining the state process.

**in.meth:** a string designating the inner optimization function: currently one of 'nlminb', 'maxNR', 'optim' (uses the 'BFGS' option), 'trust' or 'SplineEst'. The last calls `SplineEst.NewtRaph`. This function is faster if started with parameter values close to the optimal values, but may not find the global optimum as reliably without good initial values.

**out.meth:** Outer optimization function to be used, one of 'optim' (uses the 'BFGS' methods), 'nlminb', 'maxNR', 'trust' or 'subplex'. When squared error is being used, 'ProfileGN' and 'nls' can also be given. The former of these calls `Profile.GausNewt`.

**control.in:** A control object that controls the inner optimization function.

**control.out:** A control object for outer optimization function.

**active:** A vector containing indices indicating which parameters of `pars` should be estimated; it defaults to all of them.

Both `control.out` and `control.in` should give control parameters corresponding to the optimization routine being used. When there are multiple arguments used in calling those routines, they should be listed in the `control` variable. Both methods default to 'nlminb' and both control variables default to `NULL` in which case defaults from each of the methods are used.

The function returns a list with the following entries

**pars:** A vector of length  $p$  containing the optimal parameter values.

**coefs:** A  $K \times d$  matrix or array  $K \times N \times d$  array containing optimal coefficients at `pars`.

**res:** The result of the outer optimization that are specific to the optimizing function selected.

The optimization functions call lower level functions `ProfileErr` and `ProfileDP`. These functions provide, respectively, the value of the profile objective function and its derivatives with respect to parameters. They are intended as inputs into generic optimization routines and have arguments

**pars:** The vector of current parameter estimate values. Its length is determined by vector **active** and is the number of parameters being actually optimized, as opposed to the total number of parameters.

**times:** An  $n$ -vector of times

**data:** The  $n \times d$  data matrix (unreplicated) or  $n \times N \times d$  data array (replicated).

**coefs:** The starting estimates for the coefficients

**lik:** `lik` object for the data process

**proc:** `proc` object for the state process

**in.meth:** An indicator of which optimization method to use for the inner criterion. See options in `inneropt`.

**control.in:** Control parameters for optimizing coefficients.

**sgn:** Is the criterion to be maximized (1) or minimized (-1). Defaults to 1.

**active:** The indexes, or names if specified, of parameters to be estimated. Defaults to estimating all of them.

**allpars:** A list containing names of all parameters in the system, including those not being estimated; this should be a superset of **pars**.

**sumlik:** (`ProfileDP` only): the summation of the profile score vector be returned? Defaults to `TRUE`. `FALSE` can be used to generate a Newey-West estimate for the variance.

Function `ProfileErr` will create and read temporary files

- `curcoefs.tmp`,
- `optcoefs.tmp` and
- `counter.tmp`.

These are created and removed in the `outeropt`, `Profile.LS` and `profile.multinorm` functions. However, if you optimize `ProfileErr` directly, the files will not be removed. Trying to run `ProfileErr` with these files existing from previous experiments can result in errors and you should make sure to remove them before doing so.

CollocInfer function `Spline.NewtRaph` is a Newton-Raphson minimization routine to estimate the coefficients of the basis expansion for any given set of parameters. This mimics the functionality in Hooker (2006). It requires the following arguments

**coefs:** The matrix or array of coefficient values to be estimated

**times:** An  $n$ -vector of times

**data:** The data matrix or data array

**lik:** `lik` object for the data process

**proc:** `proc` object for the state process

**pars:** The current parameter estimates

**control:** Control parameters for optimizing coefficients with for each value of the parameters. These are

- reltol:** The relative change in error at which to stop. This also represents the relative gradient change at which to Stop. Defaults to  $10^{-12}$
- maxi:** Maximum number of iterations, default is 1000.
- maxtry:** The maximum number of times to half the current step before deciding the objective criterion cannot be decreased. Defaults to 10.
- trace:** Progress to report. 0 means none. 1 is on termination, 2 is by iteration.

The function returns a named list with members

**coefs:** the optimal coefficients

**g:** the gradient of the objective with respect to the coefficients

**value:** the value of the objective function

**H:** the Hessian of the objective function with respect to the coefficients

### 6.3 Function for confidence intervals: `Profile.covariance`

This function calculates a covariance estimate for the parameters, based on the Newey-West estimate outlined in Section 5. Its arguments are

**pars:** The estimated parameters

**active:** An index, or list of names, of the parameter to be estimated. Defaults to `NULL` in which case all are estimated.

**times:** Observation times.

**data:** The observed data.

**coefs:** The coefficients corresponding to the estimated parameters.

**lik:** The `lik` object used in the estimation.

**proc:** The `proc` object used in the estimation.

**in.meth:** The inner-optimization method to use, this is the same argument as in `ProfileErr`.

**control.in:** Control parameters for the inner optimization.

**eps:** The finite-difference discretization parameter, defaults to `1e-6`.

## 6.4 A special purpose optimizer for least squares: `Profile.GausNewt`

This function is most commonly called through `Profile.LS`. `Profile.GausNewt` provides a Gauss-Newton method for minimizing (6) when the likelihood is described by squared error. It mimics its namesake in Hooker (2006). It requires

**pars:** The current parameter estimates

**times:** A  $n$ -vector of times

**data:** The  $n \times d$  data matrix

**coefs:** The starting estimates for the coefficients

**lik:** `lik` object for the Markov process

**proc:** `proc` object for the Markov process

**in.meth:** An indicator of which optimization method to use for the inner criterion. See `inneropt`

**control.in:** Control parameters for optimizing coefficients with for each value of the parameters.

**active:** The indexes, or names if specified, of parameters to be estimated. Defaults to estimating all of them.

**control:** Control parameters for optimizing coefficients with for each value of the parameters. These are

**reltol:** The relative change in error at which to stop. This also represents the relative gradient change at which to Stop. Defaults to  $10^{-12}$

**maxit:** Maximum number of iterations, default is 1000.

**maxtry:** The maximum number of times to half the current step before deciding the objective criterion cannot be decreased. Defaults to 10.

**trace:** Progress to report. 0 means none. 1 is on termination, 2 is by iteration. Default is 0

This function returns a list with the following elements:

**pars:** The optimized parameter values (the full parameter vector rather than just the active parameters).

**in.res:** The result of the most recent inner optimization.

**value:** The squared error out objective criterion.

## 6.5 Gradients and Hessians for least squares: ProfileSSE

CollocInfer function `ProfileSSE` assumes that the data fitting term evaluated by object `lik` and the roughness penalty term evaluated by object `proc` are total squared error. The function calculates the necessary gradients and objective functions for either of these. It takes arguments:

**pars:** The current parameter estimates

**allpars:** A list of all parameters in the system, including those not being estimated; this should be a superset of `pars`.

**times:** An  $n$ -vector of times

**data:** The  $n \times d$  data matrix

**coefs:** The starting estimates for the coefficients

**lik:** `lik` object for the process

**proc:** `proc` object for the process

**in.meth:** An indicator of which optimization method to use for the inner criterion. These are `'BFGS'` for `'optim'` with the BFGS method, `'nlminb'` for `nlminb`, `'maxNR'` for `maxNR` in the `maxLik` package and `'house'` for `Spline.NewtRaph`.

**control.in:** Control parameters for optimizing coefficients with for each value of the parameters. Where control parameters are passed in as arguments to a function, as in `maxNR`, these should be named as members of the `control.in` list.

**active:** The indexes, or names if specified, of parameters to be estimated. Defaults to estimating all of them.

**dcdp:** the derivative of the coefficients with respect to the parameters. Defaults to `NULL` and is largely used in `Profile.GausNewt` to speed up the inner optimization.

**oldpars:** parameters from the previous iteration, defaults to `NULL` and is largely used in `Profile.GausNewt` to speed up the inner optimization.

**use.nls:** Defaults to `TRUE`, requires the same `ProfileEnv` environment to be defined as for `ProfileErr` and formats output for `nls`.

**sgn:** Is the criterion to be maximized (1) or minimized (-1). Defaults to 1.

Function output depends on `use.nls`. If `TRUE` it outputs a vector of errors with an attribute variable giving the gradient of the errors with respect to parameters. If `FALSE` it outputs a list with elements

**f:** A vector of errors for the observations to be used in a squared error criterion.

**df:** A matrix the rows of which give the derivative of each entry in **f** with respect to the parameters.

**coefs:** The optimal coefficients for the current parameters

**dcdp:** The derivative of the coefficients with respect to the current parameters.

**ProfileSSE** creates (and reads) the same temporary files as **ProfileErr**. The same warning about removing these temporary files applies.

## 6.6 Computing starting coefficient values: `FitMatchOpt`

The primary application of function `FitMatchOpt` is to provide starting coefficient estimates for state variables for which no observations are available. The function `FitMatchOpt` provides facilities to compute estimates of some state variables  $x_i$  in a process, given previous estimates of others and parameter values. Thus if in a three-variable system  $x_1$  and  $x_2$  have been observed and their data smoothed, but  $x_3$  has not been observed, this function will estimate coefficients for  $x_3(t)$ .

`FitMatchOpt` estimates unobserved states by minimizing only the second roughness penalty term in (7), that is,

$$\sum_{i=1}^d \lambda_i \int P \left[ \frac{d}{dt} \mathbf{c}_i \Phi(t), f_i(t, \Phi(t) \mathbf{C}, \boldsymbol{\theta}) \right] dt$$

with respect to the coefficients in **C** corresponding to unobserved variables, holding fixed the parameter values in **θ**. It must, therefore, be supplied with a **proc** object that defines the roughness penalty, as well as parameter values and coefficients for the observed state variable estimates.

Function `FitMatchOpt` has arguments:

**coefs:** A matrix or array containing the current estimate of the coefficients for the hidden states.

**which:** A vector containing indices of states to be estimated.

**pars:** A vector containing parameters to be used for the processes.

**proc:** The **proc** object defining the roughness penalty.

**meth:** An optional name of an R optimization function, currently one of 'nlminb', 'MaxNR', 'optim' or 'trust'.

**control:]** An optional control object for optimization function.

**control** and **meth** work exactly as the counterparts in `inneropt`.

Function `FitMatchOpt` returns a named list with members:

**coefs:** A matrix or array containing the estimated coefficients for the hidden states.

**res:** A list containing summaries of the optimization that are returned by the specific optimization function that was used.

We can illustrate the use of `FitMatchOpt` for the FitzHugh-Nagumo equations, using the code in Section 3, by estimating coefficients for variable  $R$  on the basis of observations on only  $V$ . Let us assume that the second column in the data matrix `FHN.data` contains all NA's, indicating that the recovery variable  $R$  has not been observed. In this code we set up a new coefficient array with the first column containing coefficients for variable  $V$  estimated by smoothing only this variable, and with the second column zeros, our initial coefficient values for the optimization process.

```
FHN.Vfd      = smooth.basis(times, FHN.data[,1] FHN.basis)$fd
FHN.Vcoefs   = FHN.Vfd$coefs
FHN.VRcoefs  = cbind(FHN.Vcoefs,matrix(0,FHN.nbasis,1))
```

Here is code for defining the `lik` and `proc` objects for the FitzHugh-Nagumo equations for the least squares case using function `SSEproc` described in Section 4.3.3.

```
FHN.proc= make.SSEproc()
```

Now the optimization is carried out using an initial guess at parameter values in vector `FHN.pars0`, and we then extract the estimated coefficients for variable  $R$ .

```
res = FitMatchOpt(FHN.VRcoefs,which=2,FHN.pars0,FHN.proc)
FHN.VRcoefs[,2] = res$coefs
```

At this point, we are now ready to proceed to the use of functions `inneropt` and `outeropt`, as described in Section 6.

It should not be missed that the use of `FitMatchOpt` does require that we have a reasonable set of initial values for the parameter vector  $\theta$ , supplied in argument `pars` in this code. See function `ParsMatchOpt` for a method for estimating these initial parameter values if one is in the happy situation of having observations for all the state variables. Unfortunately, there is no easy way to get good initial estimates of both  $\mathbf{C}$  and  $\theta$  at the same time. Whether one should first turn to using `FitMatchOpt` and then `ParsMatchOpt`, or vice versa, will depend on the complexity of the differential equation and the nature of the data that one has available.

As was the case for the smoothing problem, the optimization that takes place in `FitMatchOpt` relies on functions to define the objective and derivatives: `FitMatchErr` returns an objective function value calculated as the value of `proc` while `FitMatchDC` returns the derivative and `FitMatchDC2` returns the Hessian with respect to the components that are being estimated. They all take arguments:

**coefs:** The estimated coefficients for the components being estimated.

**allcoefs:** The  $K \times d$  matrix or  $K \times N \times d$  array of coefficients. This is used to extract the coefficients for the components that are held fixed. The coefficients for the components being estimated are ignored.

**which:** A vector giving the indexes of the components being estimated.

**pars:** The parameters to be used in the system.

**proc:** The `proc` object for the system in question.

**sgn:** a variable taking values +1 or -1 indicating if the objective function should be maximized (-1) or minimized (+1). Defaults to +1.

## 6.7 Estimating parameters given $\mathbf{x}$ : `ParsMatchOpt`

Function `ParsMatchOpt` minimizes only the second roughness penalty term in (7), that is,

$$\sum_{i=1}^d \lambda_i \int P \left[ \frac{d}{dt} \mathbf{c}_i \Phi(t), f_i(t, \Phi(t) \mathbf{C}, \boldsymbol{\theta}) \right] dt$$

with respect to the parameters in  $\boldsymbol{\theta}$  for a fixed set of coefficients in  $\mathbf{C}$ . This is the “gradient matching” problem that is also treated by principal differential analysis in Ramsay and Silverman (2005). When all the state variables are observed, so that  $\mathbf{C}$  can be estimated by direct smoothing, this function can be used to provide useful initial values  $\hat{\boldsymbol{\theta}}$  for the parameters.

`ParsMatchOpt` requires arguments:

**pars:** A vector of initial values of parameters in  $\boldsymbol{\theta}$  to be estimated.

**coefs:** A matrix or array containing the current coefficient estimates defining the state variables.

**proc:** A `proc` object defining the fit of state processes to the differential equation.

**active:** A vector of indices indicating which parameters of `allpar` should be estimated; defaults to all of them.

**allpars:** A vector containing all of the parameters, the assignment `allpar[active]=pars` is made initially.

**sgn:** Is the minimizing (1) or maximizing (0)?

**meth:** The optimization function: currently one of `'nlminb'`, `'MaxNR'`, `'BFGS'` or `'trust'`.

**control:** Control object for optimization function.

`ParsMatchOpt` returns a list with elements

**pars:** The optimized parameter values.

**res:** The output of the optimization routine.

For the FitzHugh-Nagumo equations, assuming that we have a full coefficient matrix `FHN.VRcoefs` containing coefficients for both the  $V$  and  $R$  state variables, we can estimate parameter values with the code

```
res = ParsMatchOpt(pars=pars, coefs=FHN.VRcoefs, FHN.proc)
FHN.pars = res$pars
```

As above in the illustration of the use of `FitMatchOpt`, we can now proceed to the final optimization phase with these parameter estimates.

The optimization routines refer to functions `ParsMatchErr` and `ParsMatchDP`. These provide the value and gradient for a fixed set of coefficients. They require

**pars:** The current estimates of the parameters

**coefs:** the coefficients (held fixed)

**proc:** the `proc` object for the state process

**active:** The indexes, or names if specified, of parameters to be estimated. Defaults to estimating all of them.

**allpars:** A list of all parameters in the system, including those not being estimated; this should be a superset of **pars**.

**sgn:** a variable taking values `+1` or `-1` indicating if the objective function should be maximized (`-1`) or minimized (`+1`). Defaults to `+1`.

## 7 More functions for least squares estimation

Ramsay et al. (2007) describe estimation methods for direct measurements of a system using squared error for both `lik` and `proc`. In particular, Hooker (2006) describes software that interfaces with the Matlab `fda` package. These functions provide equivalent functionality for the `fda` library in R. There are equivalent functions for multivariate normal distributions.

Due to the interface with the `fda` library, some standard conventions are changed when repeated time-series observations are given. We assume that the dimensions of observations (and consequently of co-efficients) describe

1. time points
2. replicates
3. variables

that is, replicated time series are included as the second dimension. This fits the formalism of the `fda` library. Note that in order to use these functions, all replicates must have the same observation times and use the same collocation basis. If this is not the case, they can be incorporated manually, as described above. If the data are given as a matrix, it is assumed that only one replicate is present.

### 7.1 Setting up `lik` and `proc` objects: `LS.setup`

This function calculates and returns `lik` and `proc` objects given some inputs, possibly including a functional data object. A fair amount of argument parsing is done here, so you should read the way various arguments are handled carefully.

**pars:** A set of starting parameters, should be named.

**coefs:** An initial set of co-efficients. If there are replicates, this should be a three dimensional array with the second dimension giving replicates, and the third indexing state vectors. If a two-dimensional array is given, it is assumed that there is only one replicate. The `dimnames` attribute of `coefs` should give the state names as they are referred to in the right hand side of the differential equation. `coefs` defaults to `NULL`, in which case these are taken from `fd.obj`.

**fn:** One of

- A list of functions as specified in `proc$more` for `SSEproc`.
- A single function giving the right hand side of the differential equation. In this case, a finite difference routine is used to estimate derivatives of the right hand side numerically.

**basisvals:** One of

- A B-spline basis object (see the `fda` library) giving the collocation basis.
- A list with elements

**bvals.obs:** the basis evaluated at the observation times  
**bvals:** the basis evaluated at collocation points  
**dbvals:** the derivative of the basis evaluated at collocation points  
**qpts:** a vector giving quadrature points  
**qwts:** a vector of quadrature weights. If this is not specified, they are all set to 1.

- NULL (default) in which case the basis is taken from `fd.object`.
- A matrix of values – for discrete-time systems only; defaults to an identity matrix of dimension the number of observations if `discrete` and `basisvals` is left as null.

**lambda:** A vector of  $\lambda$ 's, one for each state variable. If given as a singleton, it is expanded so that all the  $\lambda_i$  are the same.

**fd.obj:** A functional data object giving a smooth for the system. If this is present, it over-rides `coefs` and `basisvals` replacing them with `fd.obj$coefs` and `fd.obj$basis`. State variables are taken from the `fd.obj$fdnames`.

**more:** Additional inputs into `fn` as would normally be given.

**weights:** A matrix of observation weights. Defaults to NULL in which case these are assumed to be all 1.

**times:** A matrix of observation times. Defaults to NULL. If `fd.obj` is given, or if `basisvals` is a basis object, this must be specified.

**quadrature:** If `basisvals` is a basis object or `fd.obj` is present, otherwise ignored. A list giving the quadrature scheme. In particular it should provide

**qpts:** a vector giving quadrature points  
**qwts:** a vector of quadrature weights. If this is not specified, they are all set to 1.

If NULL (default) `qpts` is set to be midpoints between knots and `qwts` is set to be all ones.

**eps:** Size of  $h$  for finite difference approximations. Defaults to `1e-6`.

**posproc:** Is the system always positive? In this case,  $\mathbf{x}(t)$  is represented by an exponentiated basis expansion  $\exp(\Phi(t)\mathbf{C})$ . It uses `make.logtrans` in the `proc` object; see Section 8.

**poslik:** Should the trajectory be exponentiated before being compared to the data? If this is set to 1, we use `make.exp` in the `lik` object.

**discrete:** Is the system discrete-time? In this case the definitions of `basisvals` are changed to assume a difference equation.

Note that state and parameter names, if used in `fn`, are taken from `pars` and `coefs` (or `fd.obj$fdnames`) and are otherwise assumed to be `NULL`.

The function returns a list with elements

`lik`: A `lik` object using `SSElik()`.

`proc`: A `proc` object using `SSEproc()` and `fn` specifying the right hand side of a differential equation.

`coefs`: The coefficients, which are extracted from `fd.obj` if given, and re-formatted into a matrix.

## 7.2 Smoothing the data given parameters: `Smooth.LS`

This function creates `lik` and `proc` objects using `LS.setup` and runs the inner optimization. It requires the following inputs:

`fn`: As in `LS.setup`

`data`: A data array. This should be three dimensional if there are replicated time series, with the second dimension indicating the replicate number. If given as an array, one replicate is assumed.

`times`: A matrix of observation times.

`pars`: Initial parameters, should be named if names are used as indices in `fn`.

`coefs`: As in `LS.setup`.

`basisvals`: As in `LS.setup`.

`lambda`: As in `LS.setup`.

`fd.obj`: As in `LS.setup`.

`more`: Additional inputs into `fn` as would normally be given.

`weights`: As in `LS.setup`.

`in.meth`: As in `ProfileSSE`.

`control.in`: As in `ProfileSSE`.

`quadrature`: As in `LS.setup`.

`eps`: Size of  $h$  for finite difference approximations. Defaults to `1e-6`.

`posproc`: Is the system always positive? In this case,  $\mathbf{x}(t)$  is represented by an exponentiated basis expansion  $\exp(\Phi(t)\mathbf{C})$ . It uses `make.logtrans` in the `proc` object; see Section 8.

`poslik`: Should the trajectory be exponentiated before being compared to the data? If this is set to 1, we use `make.exp` in the `lik` object.

**discrete:** Is the system discrete-time? In this case the definitions of `basisvals` are changed to assume a difference equation.

After smoothing using the optimization routine given in `in.meth`, returns a list with elements

**lik:** A `lik` object using `SSElik()`.

**proc:** A `proc` object using `SSEproc()` and `fn` specifying the right hand side of a differential equation.

**coefs:** The optimized coefficients, given in the same format as the input. If `fd.obj` is input, this is omitted and an element `fd` is returned containing a functional data object using the optimized coefficients.

### 7.3 Least squares generalized profiling: `Profile.LS`

This function carries out the full profile estimate of parameters, given similar inputs as above. In particular, it requires

**fn:** As in `LS.setup`

**data:** A data array. This should be three dimensional if there are replicated time series, with the second dimension indicating the replicate number. If given as an array, one replicate is assumed.

**times:** A matrix of observation times.

**pars:** Initial parameters, should be named if names are used as indices in `fn`.

**coefs:** As in `LS.setup`.

**basisvals:** As in `LS.setup`.

**lambda:** As in `LS.setup`.

**fd.obj:** As in `LS.setup`.

**more:** Additional inputs into `fn` as would normally be given.

**weights:** As in `LS.setup`.

**in.meth:** As in `ProfileSSE`.

**out.meth:** An optimization routine for the outer optimisation. This can be either `'house'` in which case `Profile.GausNewt` is called, or `'nls'` in which case `nls` is used.

**control.in:** As in `ProfileSSE`.

**control.out:** A control list for the outer optimization. See `Profile.GausNewt` or `nls` for details.

**quadrature:** As in `LS.setup`.

**eps:** Size of  $\epsilon$  for finite difference approximations. Defaults to `1e-6`.

**posproc:** Is the system always positive? In this case,  $\mathbf{x}(t)$  is represented by an exponentiated basis expansion  $\exp(\Phi(t)\mathbf{C})$ . It uses `make.logtrans` in the `proc` object; see Section 8.

**poslik:** Should the trajectory be exponentiated before being compared to the data? If this is set to 1, we use `make.exp` in the `lik` object.

**discrete:** Is the system discrete-time? In this case the definitions of `basisvals` are changed to assume a difference equation.

The function returns:

**pars:** The optimized parameters

**res:** The object returned from the outer optimization (see `Profile.GausNewt` or `nls` as appropriate).

**lik:** A `lik` object using `SSElik()`.

**proc:** A `proc` object using `SSEproc()` and `fn` specifying the right hand side of a differential equation.

**coefs:** The optimized coefficients, given in the same format as the input. If `fd.obj` is input, this is omitted and an element `fd` is returned containing a functional data object using the optimized coefficients.

Equivalent functions to those above are given by functions `multinorm.setup`, `Smooth.multinorm` and `Profile.multinorm`. These have exactly the same arguments as their `sse` counterparts with the single exception that `lambda` is replaced by `var`.

Currently, `var` should be a 2-vector specifying the observation variance and the process variance. However, further options are being planned.

## 8 Positive State Vectors

In many systems the state vector is known to be positive: there is no such thing, for example, as a population of -50,000 fish. Enforcing this condition can be problematic for basis-expansion systems. Instead, it is often useful to represent the state on the log scale. In the case of deterministic ordinary differential equations, the relationship

$$\frac{d}{dt}\mathbf{x}(t) = \mathbf{f}(\mathbf{x}, \theta) \Leftrightarrow \frac{d}{dt}\mathbf{z}(t) = e^{-\mathbf{z}(t)}\mathbf{f}(e^{\mathbf{z}(t)}, \theta) \quad (11)$$

holds for  $\mathbf{z}(t) = \log \mathbf{x}(t)$  and can improve the conditioning of an ODE. More generally, representing  $\mathbf{x}(t) = \exp\{\Phi(t)C\}$  ensures that the state vector remains positive. The functions below provide methods of defining `lik` and `proc` objects and their derivatives using exponentiated basis expansions.

### 8.1 Utility Functions

All the functions below supplant `lik`, `proc` or other objects that are then passed into the corresponding `more` slot. These all assume that the state is represented on the log scale but that the dynamics has not been so transformed.

#### 8.1.1 `logstate.lik`

Changes a `lik` object to use the exponential of the basis expansion as the state. `make.logstate.lik()` creates a `lik` object with entries `logstate.lik.fun`, `logstate.lik.dfdx`, `logstate.lik.dfdp`, `logstate.lik.d2fdx2`, `logstate.lik.d2fdxdp`, `more`. Here `more` should be the `lik` object that is desired to be used with the exponential of the estimated state.

#### 8.1.2 `exp.Cproc`

`make.exp.Cproc` creates a `proc` object that uses an exponentiated state, treating it as a continuous time process as in `Cproc`. `more` should contain the same elements as are needed for `Cproc`.

#### 8.1.3 `exp.Dproc`

`make.exp.Dproc()` creates a `proc` object for the exponentiated state, treating it as a discrete time process as in `Dproc`. `more` should be the same as would be used with `Dproc`.

#### 8.1.4 `logtrans.ode`

Transforms the right hand side of an ordinary differential equation (and its derivatives) to the right hand side of the log state via the transformation above. `make.logstrans.ode()` creates a list that can be used as the `fn` objects in `SSEproc` for example. `more$fn` is required to be the right hand side function that is appropriate for the un-logged state.

## 9 Estimation for the FitzHugh-Nagumo system

The FitzHugh-Nagumo equations provided a test-bed for the original Matlab package described in Hooker (2006). They are given by a two-variable differential equation:

$$\begin{aligned}\frac{d}{dt}V &= c(V - V^3/3 + R) \\ \frac{d}{dt}R &= (V - a + bR)/c.\end{aligned}$$

These equations are intended to capture the essential dynamic properties neural firing behavior and may be regarded as a simplification of the Hodgkin-Huxley equations. Intuitively,  $V$  represents the voltage across the membrane of a neuron, and  $R$  is a sum of “recovery” ion currents. The problem will be to estimate parameters  $a$ ,  $b$  and  $c$ .

We first illustrate the use of CollocInfer for this system with data from a single replication, and then show its use for replicated observations.

### 9.1 Unreplicated Data

First we need to create some data. Vector `time` contains 41 equally spaced observation times spanning 0 to 20 time units. Vector `pars` contains the true values for parameters  $a$ ,  $b$  and  $c$  in the equation, and also letters used as labels for the parameter values. Vector `x0` defines the initial values at time 0 of the two variables in the system, and also two letters labeling the variables.

```
times = seq(0,20,0.5)
pars = c(0.2,0.2,3)
names(pars) = c('a','b','c')
x0 = c(-1,1)
names(x0) = c('V','R')
```

The function `make.fhn()`, contained in the CollocInfer package, sets up the named list `fhn` for the right side of the equation. This list in turn contains the functions that evaluate the right side and four of its partial derivatives. We need list `fhn` at this point because one of these functions, `fhn$fn.ode`, is used by the differential equation approximating function `lsoda` that we will use to approximate the population or true values of the differential equation solution at the times in `times`.

```
fhn = make.fhn()
```

Now we can generate some noisy observations by first computing the errorless values (well, only with the tiny errors that `lsoda` inevitably produces), and then add some normally distributed random noise with standard deviation 0.05.

```
y = lsoda(x0,times,fhn$fn.ode,pars)
y = y[,2:3]
data = y + 0.05*array(rnorm(82),2)
```

Alternatively, the object `FhHdata` in the package contains data produced by a stochastic version of the FitzHugh-Nagumo equations, making a useful test case for an inexact differential equation.

We now define basis functions for representing the state functions  $V$  and  $R$ :

```
knots = seq(0,20,0.2)
norder = 3
nbasis = length(knots) + norder - 2
range = c(0,20)
bbasis = create.bspline.basis(range=range, nbasis=nbasis, norder=norder,
                             breaks=knots)
```

Initial values for coefficients will be obtained by smoothing the noisy data

```
fd.data = array(data,c(nrow(data),1,ncol(data)))
DEfd     = data2fd(fd.data, times, bbasis,
                  fdnames=list(NULL,NULL,varnames) )
coefs    = DEfd$coefs[,1,]
colnames(coefs) = varnames
```

We also need some lists containing values that control various optimization functions.

```
control=list()
control$trace = 0
control$maxit = 1000
control$maxtry = 10
control$reltol = 1e-6
control$meth = "BFGS"

control.in = control
control.in$reltol = 1e-12
control.in$print.level = 0
control.in$iterlim = 1000

control.out = control
control.out$trace = 2
```

In order to perform profiled estimation using error sum of squares measures, we can call function `Profile.LS`, which automatically takes care of setting up the `lik` and `proc` objects for us.

```
lambda = c(10000,10000)
res0 = Profile.LS(fhn,data,times,pars,coefs=coefs,basisvals=bbasis,
                 lambda=lambda,in.meth='nlsminb',out.meth='ProfileGN',
                 control.in=control.in,control.out=control.out)
```

It's real easy to make mistakes in setting up the functions such as those in the named list `fhn`, and especially for the functions that calculate partial derivatives.

In the early stages of an analysis, where computational speed and accuracy may be secondary considerations, the use of finite differences to compute these partial derivatives may be very handy. Here we illustrate the same analysis using finite differences. This is activated by replacing the named list `fhn` as argument by a reference to only the right side evaluation function `fn`.

```
res1 = Profile.LS(fhn$fn, data, times, pars=spars, coefs=coefs,
                 lambda=lambda, in.meth='nlminb', out.meth='nls',
                 control.in=control.in, control.out=control.out)
```

A more sophisticated choice of fitting criterion involves assuming that the noisy data arise from a multivariate normal distribution. Function `Profile.mulinorm()` implements this.

```
res2 = Profile.mulinorm(fhn, data, times, pars=spars, coefs=coefs,
                       basisvals=bbasis, var=var,
                       in.meth='nlminb', out.meth='nlminb')
```

But if we want to work with `lik` and `proc` objects, we can call function `LS.setup` to these up explicitly.

```
profile.obj = LS.setup(pars=spars, fn=fhn, basisvals=bbasis,
                      lambda=10000, times=times, fd.obj=DEfd)
lik = profile.obj$lik
proc = profile.obj$proc
```

However, we can also bypass the automatic features of function `profile.obj`. For example, we might need to manually define quadrature points, weights and knots.

```
qpts = knots
qwts = rep(1/length(knots),length(knots))
qwts = qwts %*% t(lambda)
weights = array(1,dim(data))
```

Now manually define the `lik` object as squared error from the values of the system:

```
likmore = make.id()
likmore$weights = weights
lik = make.SSElik()
lik$more = likmore
lik$bvals = eval.basis(times,bbasis)
```

Object `proc` is also squared error, defined manually by

```
procmore = make.fhn()
procmore$weights = qwts
procmore$qpts = qpts
procmore$names = varnames
procmore$parnames = parnames
```

```

proc = make.SSEproc()
proc$more = procmore
proc$bvals = list(bvals = eval.basis(procmore$qpts,bbasis,0),
                 dbvals = eval.basis(procmore$qpts,bbasis,1))

```

Now we can try some optimization. We'll start off with a perturbed set of parameters:

```
spars = c(0.3,0.1,2)
```

Start with the inner optimization, using R optimization function `nlminb` to optimize:

```

res0 = inneropt(coefs, times=times, data=data, lik=lik, proc=proc,
               pars=spars, in.meth='nlminb', control.in=control.out)
ncoefs = res0$coefs

```

Since we're using squared error, we can make use of R nonlinear least squares optimizer `nls`

```

res1 = outeropt(data=data, times=times, pars=pars, coefs=coefs,
               lik=lik, proc=proc, in.meth="nlminb", out.meth="nls",
               control.in=control.in, control.out=control.out)

```

More generally, we can use `nlminb` again

```

res2 = outeropt(data=data, times=times, pars=pars, coefs=coefs,
               lik=lik, proc=proc,
               in.meth="nlminb", out.meth="nlminb",
               control.in=control.in, control.out=control.out)

```

For squared error, a Newey-West based variance estimate can be calculated from

```

Profile.covariance(pars=res1$pars, times=times, data=data,
                  coefs=res1$coefs, lik=lik, proc=proc)

```

## 9.2 Replicated Observations

In order to demonstrate replicated observations, we make use of another set of data generated at different initial conditions. We then need concatenate these observations in time, and create new values for `bvals` and `weights`. The function `diag.block` from the `simex` package is used below, but there are several packages in R that provide block-diagonal matrices.

First of all, we generate some new data and set up a data three-dimensional array for two replications, so that the second dimension has length 2 corresponding to the number of replications, and the third dimension also has length 2, but corresponding to the number of variables.

```

data2 = array(0,c(401,2,2))
data2[,1,] = y + 0.05*array(rnorm(82),2)
data2[,2,] = y + 0.05*array(rnorm(82),2)

```

Now, we'll use a least-squares smooth for initial coefficient estimates and then run profiling

```
DEfd2 = data2fd(fd.data2, times, bbasis,  
               fdnames=list(NULL,NULL,varnames))  
res3 = Profile.LS(fhn, data=data2, times=times, pars=pars,  
                 coefs=coefs, lambda=lambda, out.meth='nls',  
                 control.in=control.in, control.out=control.out)
```

The `setup` functions will work analogously.

## 10 Estimation for the groundwater system

A mudslide in a developed area in north Vancouver in 2003 claimed the lives of two people. The slide was caused, as many are, by the groundwater level rising after a series of heavy rainfalls to lubricate a boundary between two soil structures.

The city responded by contracting with a soil engineering company to install sensors that would continuously monitor the groundwater level and to provide an early warning system that would offer about six hours warning when a dangerous level was considered to be imminent. It takes about three hours for a rainfall to percolate through the trees and surface soil into the groundwater zone. In the data that are analyzed here, rainfalls are recorded and made available as hourly totals.

Soil structures work as a buffer that tends to distribute a sudden large rainfall over a longer period of time, so that groundwater does not suddenly rise in response, but rather tends to reach a new level in roughly an exponential fashion. Consequently, we proposed to the soil engineers that contacted us the following simple first order differential equation

$$\frac{dG}{dt} = -\beta(t)G(t) + \alpha(t)R(t - 3) \quad (12)$$

where  $G(t)$  is the groundwater level measured in metres above sea level,  $R(t)$  is the hourly total rainfall in millimetres, and time  $t$  is measured in hours.

Parameter  $\beta$  measures the speed with which groundwater responds to a rainfall event, and if the equation holds, a new level is effectively reached in  $4/\beta$  time units. It may be, as we indicate, that  $\beta$  should be allowed to vary slowly with time because of the fact that different subsoil structures percolate the water down through themselves at different rates. Parameter  $\alpha$  measures the size of the impact of a unit of rainfall on the rate of change of groundwater, and it, too, may vary over the observation interval.

Even though the differential equation is rather elementary, we include this problem in the manual for two reasons. First, it is an example of how an input function can be included in a model. Second, the nature of this input function is particularly troublesome for most classical differential equation software because it is discontinuous in nature with a large number of discontinuities. In fact, an analytic solution to the equation can be obtained, but is itself almost useless when used in a conventional nonlinear least squares routine because of the need to cope with its multiple discontinuities.

First we load the two sets of data; rainfall has been lagged by three hours

```
data(NSdata)

yobs = NSgroundwater
zobs = NSrainfall
tobs = NSTimes

N = length(tobs)
```

Now we want to construct a functional data object for rainfall. We use a step function constructed from an order 1 B-spline basis with a knot at each hour boundary.

```
rangeval = c(0,N-1)
norder = 1
nbasis = N + norder - 2
rainbasis = create.bspline.basis(rangeval, nbasis, norder)
rainfd = smooth.basis(tobs, zobs, rainbasis)$fd
```

We also need a basis object for representing groundwater as a functional data object, and we set up an order 3 B-spline basis with knots at the centers of each hour.

```
knots = c(rangeval[1],
          seq(rangeval[1]+0.5, rangeval[2]-0.5, len=N-1),
          rangeval[2])
norder = 3
nbasis = length(breaks) + norder
basisobj = create.bspline.basis(rangeval, nbasis, norder, knots)
```

Our first analysis will use a constant basis for both  $\beta$  and  $\alpha$ . This will provide a benchmark against which we can measure the improvement in fit when we allow these parameters to vary over time.

We now set up the two functional data objects for the parameters and store them, along with rainfall, in the `more` list object.

```
conbasis = create.constant.basis(rangeval)
betabasis = conbasis
alphabasis = conbasis
more = vector("list",0)
more$betabasis = betabasis
more$alphabasis = alphabasis
more$rainfd = rainfd
```

The named list containing the functions for evaluating the right side of the equation and its partial derivatives is set here.

```
NSfn = make.NS()
```

The next task is to set up some initial values, one set for the coefficients of the basis function expansion of groundwater, and other for the two parameters that define  $\beta$  and  $\alpha$ . The coefficient initial values are set up by smoothing the groundwater data at a light level.

```
lambdaDE = 1e0 # A good value for the initial analysis.
penorder = 1 # The penalty is first order
GfdPar = fdPar(basisobj,penorder,lambdaDE)
DEfd0 = smooth.basis(tobs, yobs, GfdPar)$fd
coefs0 = DEfd0$coefs
```

The parameter initial values are simply zeros.

```
pars0 = matrix(0, nbetabasis + nalphabasis, 1)
```

These commands set up values for the convergence criterion and output level for R function `nls`.

```
control.out = list()
control.out$trace = 1
control.out$tol = 1e-6
```

Now we're ready to roll for our first analysis of the data, where we use error sum of squares measures for both the `lik` and `proc` objects. We use the same smoothing level for this analysis as we did for the initial smooth.

```
res1 = Profile.LS(NSfn, yobs, tobs, pars0, coefs0, basisobj, lambda,
  more = more, out.meth='ProfileGN', control.out=control.out)
```

These commands display the parameter values and set up a functional data object for groundwater resulting from this analysis.

```
res1$pars
DEfd1 = fd(res1$coefs, basisobj)
```

The fit to the data from this analysis is excellent, but the light level of  $\lambda$  that has been used means that the fitting function will not satisfy the differential equation at all well. We, of course, want to see how a very near solution would do to fitting the data.

For this model, we will run into trouble if we try to run the differential equation solver `lsoda` included with the `CollocInfer` package because it assumes a smooth solution, and our solution is anything but due to the discontinuous nature of the rainfall input. We finesse this problem by increasing  $\log_{10}(\lambda)$  in five steps of size 2, each time using as initial estimates for the coefficients and parameters the estimated values obtained on the previous steps. This strategy may sound cumbersome relative to the idea of increasing  $\lambda$  directly to  $10^8$ , but is much safer because of the increasing complexity of the objective function for higher smoothing parameter levels.

Here's how the first step goes:

```
lambda = 1e2
res2 = Profile.LS(NSfn, yobs, tobs, res1$pars, res1$coefs, basisobj,
  lambda, more = more, out.meth='ProfileGN',
  control.out=control.out)
DEfd2 = fd(res2$coefs, basisobj)
```

Now we can compare the fits. This code compares the first fit defined by functional data object `DEfd1` to `DEfd5` for the final step.

```
plotfit.fd(yobs, tobs, DEfd1,
  xlab="Time (hours)", ylab="Groundwater level (metres)",
  title="")
lines(DEfd5, lty=2)
```

## 11 The Hénon Map: A Discrete System

As an example of a discrete-time system, we consider the Hénon map:

$$\begin{aligned}x_{i+1} &= 1 - ax_i^2 + y_i \\ y_{i+1} &= bx_i\end{aligned}$$

This discrete-time systems does not have a physical interpretation but has been of substantial interest as a mathematical object. Classical, chaos-generating parameters  $a$  and  $b$  are:

```
hpars = c(1.4,0.3)
```

We'll generate some data:

```
ntimes = 200
times = 1:ntimes
x = c(-1,1)
X = matrix(0,ntimes+20,2)
X[1,] = x
for(i in 2:(ntimes+20)) X[i,] = make.Henon()$ode(i,X[i-1,],hpars)
X = X[20+1:ntimes,]
Y = X + 0.05*matrix(rnorm(ntimes*2),ntimes,2)
```

From here we can call the usual smoothing functions

```
res1 = Smooth.LS(make.Henon(),data=Y,times=times,pars=hpars,coefs=Y,
                 basisvals=NULL, lambda=lambda, in.meth='nlminb',
                 control.in=control.in, pos=0, discrete=1)
```

and profiling functions. In both of these setting `discrete=1` produces basis values that correspond to a discrete-time system.

```
res2 = Profile.LS(make.Henon(), data=Y, times=times, pars=hpars,
                  coefs=Y, basisvals=NULL, lambda=lambda,
                  in.meth='nlminb', out.meth='nls',
                  control.in=control.in, control.out=control.outimes,
                  pos=0, discrete=1)
```

Setup functions for this are given by

```
profile.obj = multinorm.setup(pars=hpars,coefs=coefs,fn=make.Henon(),
                              basisvals=NULL,var=c(0.1, 0.001),times=times,discrete=1)
lik = profile.obj$lik
proc= profile.obj$proc
```

If this were to be done manually, for this case a discrete basis is the identity map:

```
basisvals = diag(rep(1,200))
```

Now lets define a process likelihood for the discrete-time system using a Gaussian transition distribution:

```
proc = make.Dproc()
proc$bvals = basisvals
proc$more = make.multinorm()
proc$more$qpts = t[1:(ntimes-1)]
proc$more$more = c(make.Henon(),make.cvar())
proc$more$more$f.more = NULL
proc$more$more$v.more =
    list(mat=1e-2*diag(rep(1,2)),sub=matrix(0,0,3))
```

and an observation likelihood is also Gaussian:

```
lik = make.multinorm()
lik$bvals = basisvals
lik$more = c(make.id(),make.cvar())
lik$more$f.more = NULL
lik$more$v.more = list(mat=diag(rep(100,2)),sub=matrix(0,0,3))
```

Optimization control variables:

```
control=list(trace = 0, maxit = 1000, maxtry = 10, reltol = 1e-6,
            meth = "BFGS")
control.in = control
control.in$reltol = 1e-12
control.out = control
control.out$trace = 2
```

Model-based smooth:

```
coefs = Y
res1 = inneropt(data=Y, times=times, pars=hpars, coefs, lik, proc,
               in.meth='nlminb', control.in)
ncoefs = matrix(res1$coefs,20)
```

Now we'll estimate some parameters

```
res2 = outeropt(data=Y, times=times, pars=hpars, coefs=coefs,
               lik=lik, proc=proc,
               in.meth="nlminb", out.meth="nlminb",
               control.in=control.in, control.out=control.out)
```

## 12 SEIR Equations and Positive State Vectors

The SEIR equations are commonly used for modeling epidemic processes. In this case, these models were taken from a study of Measles in Ontario (Hooker, Ellner, Earn, and Roditi, 2010). The SEIR equations are of the form:

$$\begin{aligned}\dot{S} &= \mu - [\beta(t)(I + v) + \nu]S \\ \dot{E} &= \beta(t)(I + v)S - (\sigma + \nu)E \\ \dot{I} &= \sigma E - (\gamma + \nu)I\end{aligned}\tag{13}$$

Here  $S$  is the number of people in a population that are susceptible to the disease,  $E$  is the number exposed and  $I$  is the number infected. There is usually an additional state

$$\dot{R} = \gamma I - \nu R$$

to represent the population of recovered (and therefore immune) individuals. However, we will assume that we only observe  $I$  and can ignore  $R$ . The parameters have the following representation

$\mu$  birth rate – treated as constant

$\beta(t)$  infection rate; this is parameterized by a constant plus sin and cos functions with period of one year

$$\beta(t) = \beta_0 + \beta_1 \sin(2\pi t) + \beta_2 \cos(2\pi t)$$

$v$  number of infective visitors

$\nu$  death rate

$\sigma$  rate of movement from Exposed to Infectious.

$\gamma$  rate of recovery from infection.

A stochastic version of (13) is given by a Gillespie process (Doob, 1945; Gillespie, 1977) in which transitions between states are given Poisson rates. The data in `SEIRdata` are generated from such a process, contaminated with multiplicative, log-normal noise:

```
data(SEIRdata)
```

```
SEIRtimes = SEIRtimes  
SEIRdata = SEIRdata
```

These are univariate. In order to make the data multivariate we will expand it for each state with `NA` in the unmeasured states. It will also be useful to use the log of the data:

```
data = cbind(matrix(NA,length(SEIRdata),2),SEIRdata)  
logdata = log(data)
```

We are also given variable names and parameters

```
SEIRvarnames = SEIRvarnames
SEIRparnames = SEIRparnames
```

```
SEIRpars = SEIRpars
```

The SEIR equations are also built in

```
SEIRfn = make.SEIR()
```

We now need to fill in a function definition for  $\beta(t)$  and its derivatives

```
beta.fun = function(t,p,more){
  return( p['b0'] + p['b1']*sin(2*pi*t) + p['b2']*cos(2*pi*t) )
}

beta.dfdp = function(t,p,more){
  dfdp = cbind(rep(1,length(t)), sin(2*pi*t), cos(2*pi*t))
  colnames(dfdp) = c('b0','b1','b2')
  return(dfdp)
}
```

These will be past to SEIRfn in a more object

```
betamore = list(beta.fun=beta.fun,
               beta.dfdp=beta.dfdp,
               beta.ind=c('b0','b1','b2'))
```

We will create a basis with knots at weekly intervals, and define a fairly large smoothing parameter.

```
rr = range(times)
knots = seq(rr[1],rr[2],1/52)
norder = 3
nbasis = length(knots)+norder-2

bbasis = create.bspline.basis(range=rr,norder=norder,n
                              basis=nbasis,breaks=knots)
```

From here we can set up the proc and lik functions

```
objs = LS.setup(SEIRpars,fn=SEIRfn,fd.obj=DEfd,more=betamore,
               data=data,times=SEIRtimes,posproc=1,poslik=0,
               names=SEIRnames,lambda=c(100,1,1))
```

```
proc = objs$proc
lik = objs$lik
```

Here, specifying `posproc=1` indicates that (13) should be transformed to model the log state variables instead of the original variables, while setting `poslik=1` indicates that we will compare the estimated log state variables to the given data rather than re-exponentiating first. That is, we are indicating we will use the log data rather than the original data. This is appropriate given log-normal multiplicative noise, it is also numerically much faster.

To get an initial starting point we will smooth the log data for  $I$  first and set all the other states to zero.

```
DEfd = smooth.basis(SEIRtimes,logdata[,3],fdPar(bbasis,1,0.1))

plotfit.fd(log(SEIRdata),SEIRtimes,DEfd$fd)

coefs = cbind(matrix(0,bbasis$nbasis,2),DEfd$fd$coefs)
DEfd = fd(coefs,bbasis)
```

The next thing to do will be to estimate the log states  $S$  and  $E$  to best match the differential equation

```
res1 = FitMatchOpt(coefs=coefs,which=1:2,proc=proc,pars=pars)
```

and we can examine the results of this by plotting them graphically

```
DEfd1 = fd(res1$coefs,bbasis)
plot(DEfd1,ylim=c(5,13))
points(SEIRtimes,logdata[,3])
```

The resulting plot is given in Figure 1. We can now run an initial smooth using the estimated coefficients as starting points.

```
res2 = inneropt(data=logdata, times=times, pars=SEIRpars,
               proc=proc, lik=lik, coefs=res$coefs)
```

And call the optimizing functions. In this case we can use the `active` input to indicate that we are only interested in fitting parameters  $i$ ,  $b_0$ ,  $b_1$  and  $b_2$

```
res3 = outeropt(data=logdata, times=times, pars=SEIRpars,
               proc=proc, lik=lik, coefs=res2$coefs,
               active=c('i', 'b0', 'b1', 'b2'))
```

Following this, we can plot the estimated trajectories

```
DEfd3 = fd(res3$coefs,bbasis)
plot(DEfd3,lwd=2,ylim=c(5,14))
```

And compare the data to the estimated trajectory

```
plotfit.fd(logdata[,3],SEIRtimes,DEfd3[3],ylab='Fit to Data')
```

As we can see in Figure 1, this looks very reasonable. It is also useful to look at the discrepancy between the estimated trajectory and the differential equation. To do this we first evaluate the trajectory at a set of points

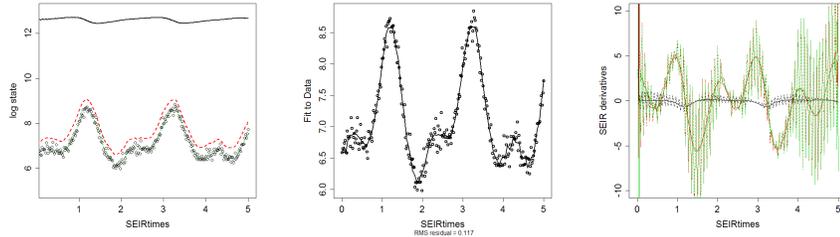


Figure 1: The results of the profile process on the SEIR data. Left: after smoothing the data for  $I$  and estimating states  $S$  and  $E$  to agree with (13). Center: a fit to data after profiling. Right: derivatives of the estimated smooth (dashed, jagged), and values of (13) for the estimated smooth (solid, smooth).

```
traj = eval.fd(SEIRtimes,DEfd3)
colnames(traj) = SEIRvarnames
```

We can then look at both the derivative of that trajectory and the value predicted by the right hand side of (13)

```
dtraj = eval.fd(SEIRtimes,DEfd3,1)
ftraj = proc$more$fn(SEIRtimes,traj,res3$pars,proc$more$more)
```

Plotting these together (see Figure 1 we see that the match is not exact, but it is fairly good

```
matplot(SEIRtimes,dtraj,type='l',lty=1,ylim =c(-10,10),
        ylab='SEIR derivatives')
matplot(SEIRtimes,ftraj,type='l',lty=2,add=TRUE)
```

An alternative strategy here is to fit the data directly (without logging), but keep the model for the trajectory on the log scale. If we do this, setting `poslik=1` indicates that the estimated trajectory should be exponentiated before being compared to the data. The entire fitting sequence is given below

```
objs2 = LS.setup(SEIRpars,fn=SEIRfn,fd.obj=DEfd,more=betamore,data=data,
                times=SEIRtimes,posproc=1,poslik=1,names=SEIRvarnames,
                lambda=c(100,1,1))
```

```
lik2 = objs2$lik
proc2 = objs2$proc
```

```
res2 = inneropt(data=data,times=SEIRtimes,pars=SEIRpars,
                proc=proc2,lik=lik2,coefs=res2$coefs)
```

```
res3 = outeropt(data=data,times=SEIRtimes,pars=SEIRpars,
```

```

proc=proc2,lik=lik2,coefs=res2$coefs,
active=c('i','b0','b1','b2'))

```

Note that this can take some hours to run.

Below, we briefly demonstrate the manual set-up of the `lik` and `proc` objects. First matrices giving the evaluation of the basis functions at the observation and quadrature times must be produced

```

qpts = 0.5*(knots[1:(length(knots)-1)] + knots[2:length(knots)])

bvals.obs = Matrix(eval.basis(times,bbasis),sparse=TRUE)

bvals = list(bvals = Matrix(eval.basis(qpts,bbasis),sparse=TRUE),
            dbvals = Matrix(eval.basis(qpts,bbasis,1),sparse=TRUE))

```

In order to use the log trajectory, we can refer `proc` to `lotrans.ode` which then calls the SEIR functions. This essentially adds an extra layer of “more” to the object

```

lsproc = sproc
lsproc$more = make.logtrans()
lsproc$more$more = make.SEIR()
lsproc$more$more$more = betamore
lsproc$more$qpts = qpts
lsproc$more$weights = matrix(1,length(qpts),3)%*%diag(c(1e2,1e0,1e0))
lsproc$more$names = SEIRnames
lsproc$more$parnames = SEIRparnames

```

This is in comparison to the `proc` object that does not take the log transformation can call the SEIR functions directly.

```

sproc = make.SSEproc()
sproc$bvals = bvals
sproc$more = make.SEIR()
sproc$more$more = betamore
sproc$more$qpts = qpts
sproc$more$weights = matrix(1,length(qpts),3)%*%diag(c(1e2,1e0,1e0))
sproc$more$names = SEIRnames
sproc$more$parnames = SEIRparnames

```

Similarly, the `make.logstate.lik` function allows us to take a standard `lik` object and use an exponentiated basis to measure the state. First we define the `lik` object as though we were making a direct comparison with the data

```

slik = make.SSElik()
slik$bvals = eval.basis(times,bbasis)
slik$more = make.id()
slik$more$weights = array(1,dim(data))
slik$more$names = SEIRnames
slik$more$parnames = SEIRparnames

```

Then we can modify this to

```
lslik = make.logstate.lik()
lslik$bvals = slik$bvals
lslik$more$weights = slik$more$weights
lslik$more = slik
lslik$more$parnames = SEIRparnames
```

We can now call these with

```
res2 = inneropt(data=logdata, times=SEIRtimes, pars=SEIRpars,
               proc=lsproc, lik=lslik, coefs=res$coefs)

res3 = outeropt(data=data, times=SEIRtimes, pars=SEIRpars,
               proc=lsproc, lik=lslik, coefs=res2$coefs,
               active=c('i', 'b0', 'b1', 'b2'))
```

Note that since we are re-exponentiating before comparing to the data, this can take a very long time.

## 13 Equations for Ecologies and Observations of Linear Combinations of States

Chemostat experiments allow ecological researchers to experimentally examine population dynamics in a strictly controlled regime. In the experiments of interest, an algae, *Chlorella vulgaris* is grown in a small (330ml) tank of water that is continuously stirred. Nitrogen, the main food for *Chlorella*, is added to the tank at a rate  $N_I(t)$  and the tank contents are evacuated and replenished with fresh water at a continuous rate  $\delta$ . Once a sufficient algal population has been established, a population of rotifers, *Brachionus calyciflorus*, is added to the chemostat. These are near-microscopic animals that feed upon the algae. A sample is then taken from the chemostat on a daily basis and the number of rotifers and algae in the sample are counted.

The data in `ChemoData` are the result of such an experiment as reported in Yoshida et al. (2003). These data are modeled with ordinary differential equations based on the state variables

1.  $N(t)$ : concentration of Nitrogen
2.  $C_i(t)$ : concentration of algal cells for different algal clones  $i = 1, 2$
3.  $B(t)$ : concentration of breeding rotifers
4.  $S(t)$ : concentration of senescent rotifers

Although the algae originate from a single culture, the models are based on a separation in real time into two different clonal types,  $C_1$  and  $C_2$ , which are required to explain the observed qualitative dynamics of the chemostat. It is this division, if it can be statistically validated, that provides evidence for real-time evolution. Additionally, rotifers are divided between breeding and senescent animals according to whether they continue to reproduce. Inference about the underlying dynamics in the Chemostat is complicated by lack of physically-observable differences between algal clones and between breeding and senescent rotifers, thus data is only available for total algal concentration,  $C_1(t) + C_2(t)$ , and total rotifer concentration  $B(t) + S(t)$ .

The evolution of these states is modeled by the following equations

$$\begin{aligned}
 \frac{dN}{dt} &= \delta(N_I(t) - N) - \frac{\rho C_1 N}{K_{C_1} + N} - \frac{\rho C_2 N}{K_{C_2} + N} \\
 \frac{dC_i}{dt} &= C_i \left[ \frac{\chi_C \rho N}{K_{C_1} + N} - \frac{p_i G(B + S)}{K_B + \max(p_1 C_1 + p_2 C_2, Q^*)} - \delta \right], \quad i = 1, 2 \\
 \frac{dB}{dt} &= B \left[ \frac{\chi_B G(p_1 C_1 + p_2 C_2)}{K_B + \max(p_1 C_1 + p_2 C_2, Q^*)} - (\delta + m + \lambda) \right] \\
 \frac{dS}{dt} &= \lambda B - (\delta + m)S.
 \end{aligned} \tag{14}$$

The equations above result in 12 parameters ( $\rho, K_{C_1}, K_{C_2}, \chi_C, p_1, p_2, K_B, Q^*, \chi_B, m, G, \lambda$ ) along with directly controllable terms  $\delta$  and  $N_I(t)$ . Some of these have established

values in the biological literature and others need to be estimated. Additionally, it is visually apparent that while the models may capture the qualitative dynamics of the system, there is considerable stochastic variation evident in the observed system that also needs to be accounted for.

We start by taking logs of the pa

```
lpars=c(ChemoPars[1:2],log(ChemoPars[3:16]))
```

Parameters  $p_2$  and  $p_1$  represent relative palatability of the two algal clones, as such only one can be estimated and we fix  $p_2 = 0$ .

```
active = c(1:5,7:16)
```

We'll choose a fairly large value of lambda.

```
lambda = rep(100,5)
```

We need some basis functions

```
rr = range(ChemoTime)
knots = seq(rr[1],rr[2],by=0.5)
bbasis = create.bspline.basis(rr,norder=4,breaks=knots)
```

We will also have to set up the basis matrices manually.

```
mids = c(min(knots),(knots[1:(length(knots)-1)] + 0.25),max(knots))

bvals.obs = eval.basis(ChemoTime,bbasis)

bvals.proc = list(bvals = eval.basis(mids,bbasis),
                  dbvals = eval.basis(mids,bbasis,1));
```

We can now set up the proc object. We will want to take a log transformation of the state here for numerical stability. In general it is better to do finite differencing *after* the log transformation rather than before it.

```
proc = make.SSEproc()           # Sum of squared errors
proc$bvals = bvals.proc         # Basis values
proc$more = make.findif.ode()   # Finite differencing
proc$more$more = list(fn=make.logtrans()$fn,eps=1e-8) # Log transform
proc$more$more$more = list(fn=chemo.fun) # ODE function
proc$more$qpts = mids           # Quadrature points
proc$more$weights = rep(1,5)*lambda # Quadrature weights
proc$more$names = ChemoVarnames # Variable names
proc$more$parnames = ChemoParnames # Parameter names
```

For the lik object we need to both represent the linear combination transform and we need to model the observation process.

First to represent the observation process, we can use the genlin functions. These produce a linear combination of the the states (they can be used in proc objects for linear systems, too).

```
temp.lik = make.SSElik()
temp.lik$more = make.genlin()
```

`genlin` requires a `more` object with two elements. The `'mat'` element gives a template for the matrix defining the linear combination. This is all zeros 2x5 in our case for the two observations from five states. The `'sub'` element specifies which elements of the parameters should be substituted into the `mat` element. `'sub'` should be a kx3 matrix, each row defines the row (1) and column (2) of `'mat'` to use and the element of the parameter vector (3) to add to it.

```
temp.lik$more$more = list(mat=matrix(0,2,5,byrow=TRUE),
  sub = matrix(c(1,2,1,1,3,1,2,4,2,2,5,2),4,3,byrow=TRUE))
temp.lik$more$weights = c(10,1)
```

Finally, we tell `CollocInfer` that the trajectories are represented on the log scale and must be exponentiated before comparing them to the data.

```
lik = make.logstate.lik()
lik$more = temp.lik
lik$bvals = bvals.obs
```

Because we don't have direct observations of any state, we'll use a starting smooth obtained from generating some ODE solutions

```
y0 = log(c(2,0.1,0.4,0.2,0.1))
names(y0) = ChemoVarnames

odetraj = lsoda(y0,1:160,func=chemo.ode,parms=lparms)

DEfd = smooth.basis(ChemoTime,odetraj[110:160,2:6],
  fdPar(bbasis,int2Lfd(2),1e-6))
C = DEfd$fd$coef
```

Now, with parameters fixed, we'll estimate coefficients.

```
res = inneropt(coefs=C,parms=lparms,times=ChemoTime,data=ChemoData,
  lik=lik,proc=proc,in.meth='optim')
```

We'll for the trajectory and also the appropriate sum of exponentiated states to compare to the data.

```
C = matrix(res$coefs,dim(C))
traj = lik$bvals%*%C
obstraj = lik$more$more$fn(ChemoTime,exp(traj),lparms,lik$more$more$more)
```

Plot these against the data

```

X11()
par(mfrow=c(2,1))
plot(obstraj[,1],type='l',ylab='Chlamy',xlab='')
points(ChemoData[,1])
plot(obstraj[,2],type='l',ylab='Brachionus',xlab='days')
points(ChemoData[,2])

```

Now we can continue with the outer optimization

```

res2 = outeropt(pars=lpars,times=ChemoTime,data=ChemoData,coef=C,
               lik=lik,proc=proc,active=active,
               in.meth='optim',out.meth='nlminb')

```

We'll extract the resulting parameters and coefficients.

```

npars = res2$npars
C = as.matrix(res2$coefs,dim(C))

```

And obtain an estimated trajectory and the exponentiated sum to compare to the data.

```

traj = lik$bvals%*%C
ptraj = lik$more$more$fn(ChemoTime,exp(traj),npars,lik$more$more$more)

```

Now we can produce a set of diagnostic plots. Firstly, a representation of the trajectory compared to the data.

```

X11()
par(mfrow=c(2,1))
plot(ChemoTime,ptraj[,1],type='l',ylab='Chlamy',xlab='')
points(ChemoTime,ChemoData[,1])
plot(ChemoTime,ptraj[,2],type='l',ylab='Brachionus',xlab='days')
points(ChemoTime,ChemoData[,2])

```

Now we'll plot both the derivative of the trajectory and the value of the differential equation right hand side at each point. This represents the fit to the model.

```

traj2 = proc$bvals$bvals%*%C
dtraj2 = proc$bvals$dbvals%*%C

colnames(traj2) = ChemoVarnames
ftraj2 = proc$more$fn(proc2$more$qpts,traj2,npars,proc$more$more)

```

```

X11()
par(mfrow=c(5,1),mai=c(0.3,0.6,0.1,0.1))
for(i in 1:5){
  plot(mids,dtraj2[,i],type='l',xlab='',ylab=ChemoVarnames[i])
  lines(mids,ftraj2[,i],col=2,lty=2)
  abline(h=0)
}
legend('topleft',legend=c('Smooth','Model'),lty=1:2,col=1:2)

```

## 14 Acknowledgements

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 Grant NSF DEB-0813743 and Federal Formula Funds Hatch Grant NYC-150446.

## References

- Cao, J. and J. O. Ramsay (2009). Linear mixed effects modeling by parameter cascading. *Journal of the American Statistical Association* 105.
- Doob, J. L. (1945). Markoff chains denumerable case. *Transactions of the American Mathematical Society* 58(3), 455-473.
- Gillespie, D. T. (1977). Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry* 81(25), 2340-2361.
- Hooker, G. (2006). Matlab functions for the profiled estimation of differential equations. Technical report, Cornell University.
- Hooker, G., S. P. Ellner, D. Earn, and L. Roditi (2010). Parameterizing state-space models for infectious disease dynamics by generalized profiling: Measles in ontario. Technical report, Cornell University.
- Newey, W. K. and K. D. West (1987). A simple, positive semi-definite, heteroskedasticity and autocorrelation consistent covariance matrix. *Econometrica* 55, 703-708.
- Ramsay, J. O., G. Hooker, D. Campbell, and J. Cao (2007). Parameter estimation in differential equations: A generalized smoothing approach. *Journal of the Royal Statistical Society, Series B (with discussion)* 65(5), 741-796.
- Ramsay, J. O., G. Hooker, and S. Graves (2009). *Functional Data Analysis with R and Matlab*. New York: Springer.
- Ramsay, J. O. and B. W. Silverman (2005). *Functional Data Analysis*. New York: Springer.
- Yoshida, T., L. E. Jones, S. P. Ellner, G. F. Fussmann, and N. G. Hairston (2003). Rapid evolution drives ecological dynamics in a predator-prey system. *Nature* 424, 303-306.